National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

University of Alberta

Transaction Management on Multidatabase Systems

by

Kenneth E. Barker

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta
Fall 1990

National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Service des thèses canadiennes

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-64782-5

Canada

University of Alberta
Edmonton

Department of Computing Science
Faculty of Science

Canada T6G 2H1

779 General Services Building, Telephone (403) 492-2860
Fax:(403) 492-1071 email: ozsu@cs.ualberta.ca, ozsu@ualtamts.bitnet

23 July 1990

To whom it may concern,

I consent to the use of the contents of the following papers that Kenneth E. Barker and I have co-authored:

"Concurrent Transaction Execution in Multidatabase Systems", In *Proc. COMPSAC'90*, Chicago, Illinois, October 1990, and

"Architectural Classification and Transaction Execution Models of Multidatabase Systems", In *Proc. International Conference in Computing and Information*, May 1990, Niagara Falls, Canada, pp. 275-279

to be used by Kenneth E. Barker in his Ph.D. dissertation. These papers form the basis of Section 1.1 and Chapter 4 of his thesis entitled "Transaction Management on Multidatabase Systems".

Sincerely yours,

M. Tamer Özsu

M. Tamer Özsu
Associate Professor

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kenneth E. Barker

TITLE OF THESIS: Transaction Management on Multidatabase Systems

DEGREE: Doctor of Philosophy

YEAR THIS DEGREE GRANTED: 1990

Permission is hereby granted to the UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed)

Permanent Address:
256 South Ridge,
Edmonton, Alberta,
Canada
T6H 4M9

Date: 25 July /90

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Transaction Management on Multidatabase Systems** submitted by **Kenneth E. Barker** in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

M. Tamer Özsu
M. Tamer Özsu
(Supervisor)

T. A. Marsland
T. Anthony Marsland

Li Yan Yuan
Li Yan Yuan

Jack Mowchenko
Jack Mowchenko

Rusinkiewicz
Marek Rusinkiewicz

Date: [illegible]

Abstract

Two components of transaction management in multidatabase systems are concurrency control and reliability. Multidatabase systems constructed from autonomous independent database managers are an alternative to homogeneous integrated distributed database systems. Multidatabase transaction management has long been of interest, but general solutions to the problems have only recently been forthcoming. Early approaches were either read-only or permitted updates to only one of the underlying databases at a time, required modification to the underlying database systems, or suggested that the DBMS scheduler be replaced with another which controlled transactions from “above”. The most pessimistic scenario is assumed in this research, where individual database managers on the multidatabase system are totally autonomous. The work contributes first by demonstrating the importance of and provide a formal model for multidatabase systems. The formalism is capable of capturing the issues related to both concurrency control and reliability. The second contribution is to provide two new concurrency control algorithms which permit global updates that span multiple databases. The correctness of these algorithms is demonstrated using an extension of the well-known serializability notion called *multidatabase serializability*. Most importantly the thesis contributes by reporting on the development of algorithms which guarantee reliability. The algorithms facilitating reliability ensure both transaction atomicity and crash recovery.

Acknowledgements

I owe a debt of gratitude to a large number of people for supporting me in the production of this thesis. First, I would like to thank my supervisor Dr. M. Tamer Özsu for so many different things that I cannot attempt to list them here. He has supported the research, even when I was not sure where it was going, by encouragement, patience, understanding and caring. I thank Tamer for teaching me about research and helping me to understand the process of accomplishing it. Two others have helped this thesis become a scholarly piece of work. I thank Dr. Li Yan Yuan for the great care he has taken with the proofs, examples and algorithms to ensure their correctness. Dr. Tony Marsland was the first person I met when I arrived at the department. Tony "... rubbed my nose in it" when it came to the English in the thesis. I believe this has made the it a *much* better peice of work. He has also helped me to write much gooder! Thanks are also due to Dr. Marek Rusinkiewicz and Dr. Jack Mowchenko for their careful review of the work and helpful comments.

Undoubtedly, Sharon E. Barker (Ph.T.) (Put Hubby Through) deserves the lion's share of the thanks. I love you deeply! You have given me two beautiful children during this time, much love and an understanding which is incomparable. This work and success is as much yours as mine.

I'd also like to thank our parents. They believed in us, even if they did not always understand why we were doing what we were doing, they were understanding and loving. They have supported us financially, emotionlly and lovingly. These are depts that cannot be paid back (except maybe the financial ones), but I can promise you that we will return the favor to your grandchildren.

Finally, those around the department who have supported both the research and myself in many different ways. Although I cannot mention all of them a few should be recognized. I'd like to thank the folks in the lab, K.L., Dave S., Rasit, Dave M., Koon, Ajit, Randy, Mei Fen, Christina *etc.*, who were caring and bright enough to kick the research hard but never did it in such a way that I could take it personally. I thank the support people at the department for providing an excellent research environment. In particular I'd like to single out Lynn K. for his support and friendship over the last four years, you've made the experience much more enjoyable.

Contents

1 Introduction **1**

1.1 A Taxonomy 3

1.2 MDS Architecture 8

1.3 Outline of Thesis 11

2 Background **12**

2.1 Fundamental Definitions 12

2.1.1 Serializability 14

2.1.2 Reliability 15

2.2 Summary of Related Research 18

3 A Formal Model of MDS **26**

3.1 Transaction Types 26

3.2 Histories 29

3.3 MDB-Serializability 33

3.4 MDS Recoverability 34

3.4.1 Local Level Recoverability 34

3.4.2 Global Recoverability 35

3.5 Summary of Assumptions 41

4 Multidatabase Concurrency Control **44**

4.1 Multidatabase Serializability Graphs . . . 46
4.2 Multidatabase Serializability Theorem . . . 49
4.3 Comparison to Other Serializability Criteria . . . 52
4.4 MDB Schedulers . . . 57
  4.4.1 Global Serial Scheduler . . . 59
  4.4.2 Correctness of the Global Serial Scheduler . . . 70
  4.4.3 Aggressive Global Serial Scheduler . . . 74
  4.4.4 Correctness of the Aggressive Global Serial Scheduler . . . 85
4.5 Concluding Remarks . . . 86

5 MDS Reliability **89**

5.1 Types of Failures . . . 90
  5.1.1 DBMS Failure . . . 91
  5.1.2 Transaction Failure . . . 92
  5.1.3 Media Failures . . . 93
5.2 MDMS Data Management Architecture . . . 94
  5.2.1 Local Data Manager Architecture . . . 94
  5.2.2 Global Recovery Manager Architecture . . . 97
  5.2.3 Global Logging . . . 99
5.3 Two phase Commit in MDSs . . . 101
  5.3.1 Overview . . . 102
  5.3.2 The Technique . . . 102
  5.3.3 Individual DBMS Requirements . . . 108
  5.3.4 Global Transaction Submission . . . 111
5.4 MDB Recovery Procedures . . . 114
  5.4.1 Modified Local Restart Process . . . 114
  5.4.2 Global Restart Process . . . 118

5.5 Correctness of Two-Phase Commit Algorithms 123

5.5.1 Termination Protocols 124

5.6 Correctness of Recovery Protocols 126

5.6.1 DBMS System Failure(s) 127

5.6.2 MDMS System Failure 128

5.6.3 DBMS(s) and MDMS System Failure 129

6 Conclusion **131**

List of Figures

1.1 DBMS Implementation Alternatives . . . 3

1.2 Components of an MDS . . . 9

1.3 MDS Architecture . . . 10

2.1 Relationship between histories that are SR, RC, ACA and ST . . . 17

3.1 Depiction of the Computational Model . . . 29

3.2 Recoverability Containment Properities of Local Histories . . . 36

3.3 Types of Local Histories Possible in a Global History which are (a) GRC, (b) AGCA, and (c) GIST . . . 38

3.4 Recoverability Containment Properties of Global Histories . . . 40

4.1 Intermediate Graph Representation . . . 47

4.2 MSG for MH Described in Examples 3.1 and 4.1 - 4.3 . . . 50

4.3 QSR and MDBSR Related to VSR and CSR . . . 53

4.4 Sample Correctness Graphs: (a) SG (b) QSG (c) MSG . . . 55

4.5 Global Serial Scheduler (Part 1) . . . 66

4.6 Global Serial Scheduler (Part 2) . . . 68

4.7 Trace of Scheduler Execution for Example 4.3 . . . 71

4.8 Aggressive Global Serial Scheduler (Part 1) . . . 79

4.9 Aggressive Global Serial Scheduler (Part 2) . . . 80

4.10 Trace of Scheduler Execution for Example 4.4 . . . 84

5.1 Generic Model of a Local Data Manager 95

5.2 Architecture of the Global Recovery Manager 98

5.3 Detailed Architecture of the Global Recovery Manager 100

5.4 State Diagram for MDB Two Phase Commit Algorithms 103

5.5 Global Logging of the Global Recovery Manager 112

5.6 Modified Local Restart Procedure Using the ORACLE Syntax 116

5.7 State Diagram for MDB Two-Phase Commit Algorithms Including Logging 125

Chapter 1

Introduction

Research and development efforts during the last decade have made distributed database systems (DDBS) a commercial reality. A number of products are on the market and more are being introduced. It is claimed that in the next ten years there will be a significant move toward distributed data managers so that centralized database managers will become an "antique curiosity" [39, page 189].

Two commonly cited advantages of distributed database systems are local autonomy and sharability of data and resources. One way to realize these advantages is to build distributed systems in a bottom-up fashion, by putting together existing centralized database managers (DBMSs). This gives rise to a *multidatabase system* (MDS), which can be defined as an interconnected collection of autonomous databases. A more precise definition will be given in the next section.

This thesis address the problem of concurrent transaction execution and the reliability of transactions in multidatabase systems. *Multidatabase serializability* is presented, which is an extension of serializability theory applicable to multidatabase systems. Multidatabase serializability is the theoretical basis used to reason about scheduling algorithms in these database environment. The thesis then describes schedulers for executing transactions concurrently in a multidatabase environment.

Reliability is comprised of two parts. First, a system is reliable if all the effects of committed transactions are reflected on the database but the effects of uncommitted or aborted ones do not appear. This is known as *transaction atomicity*. Fundamental to the discussion of transaction atomicity is the determination of the point of commitment. The multidatabase environment requires the development of a set of definitions which describe precisely what constitutes the commitment of a transaction. Second, reliability requires that in the event of a failure the system is able to recover the database to a consistent state such that every transaction terminates according to the first condition. This is known as *crash recovery*.

Most of the early work on multidatabase systems has concentrated on the heterogeneous nature of such systems and the associated problems of data and program conversion, integration, and access to multiple databases. The earlier prototype systems were either read-only or supported updates in batch mode, without any support for concurrent updates. A brief summary of existing work is provided in Chapter 2.

The thesis contributes in the following areas:

1. Provides a precise definition of the multidatabase environment and the difficulties encountered in providing general transaction processing capabilities including transaction atomicity and crash recovery.
2. Introduces a new correctness criteria for the multidatabase environment.
3. Provides two scheduling algorithms that guarantee correct execution of global update transactions while maintaining the autonomy of the underlying DBMSs.
4. Demonstrates that the scheduling algorithms are correct.
5. Proposes an implementation scheme for the well-known two-phase commitment protocol and the related termination and recovery algorithms.
6. Proves that these algorithms guarantee recoverable execution of concurrent transactions.

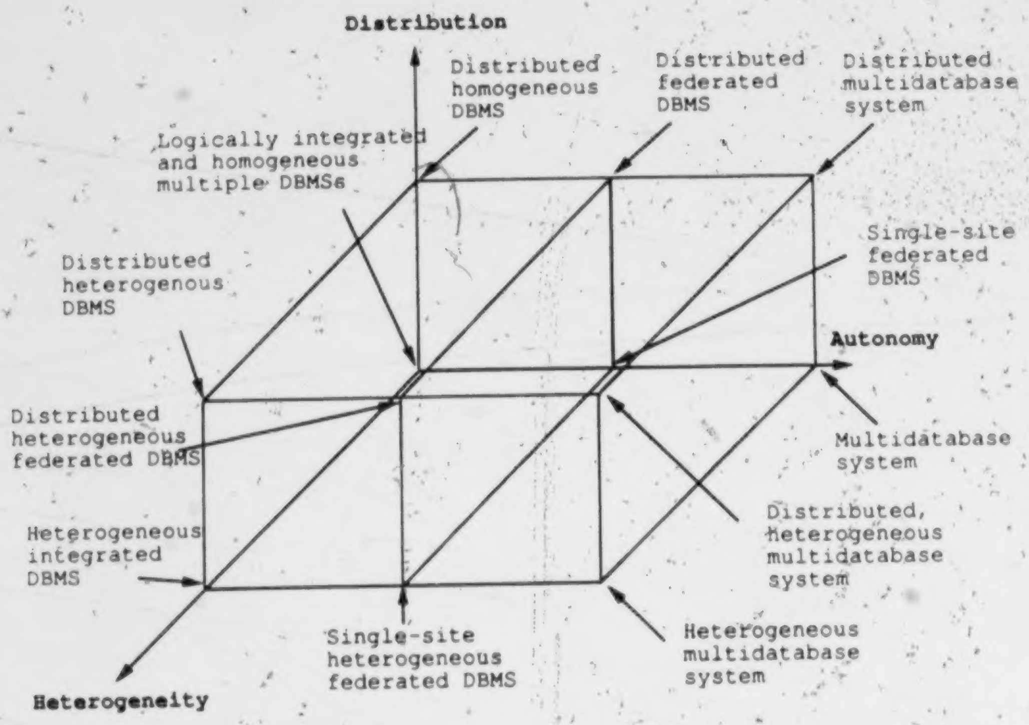

Distribution
Distributed homogeneous DBMS
Distributed federated DBMS
Distributed multidatabase system
Logically integrated and homogeneous multiple DBMSs
Single-site federated DBMS
Distributed heterogenous DBMS
Autonomy
Distributed heterogeneous federated DBMS
Multidatabase system
Distributed, heterogeneous multidatabase system
Heterogeneous integrated DBMS
Single-site heterogeneous federated DBMS
Heterogeneous multidatabase system
Heterogeneity

Figure 1.1: DBMS Implementation Alternatives

1.1 A Taxonomy

We[1] use a classification (Figure 1.1) which characterizes the systems with respect to (1) the autonomy of local systems (2) their distribution, and (3) their heterogeneity. Autonomy is the more important of these characteristics so it will be emphasized in this presentation.

Autonomy refers to the distribution of control and indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems exchange information, whether they can independently execute transactions, and whether they are modifiable. Requirements of an autonomous system have been specified in a variety of ways. For example, the following requirements are listed in Gligor and Popescu-Zeletin [22]:

---

[1] This section is published in the *Proceedings of the International Conference on Computing Information*, (ICCI '90) [33] and forms the basis of the architectural discussion in the work of Ozsu and Valduriez [34].

1. The local operations of the individual DBMSs are not affected by their participation in the multidatabase system.
2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.
3. System consistency or operation should not be compromised when individual DBMSs join or leave the multidatabase confederation.

On the other hand, Du and Elmagarmid [10] specify the dimensions of autonomy as:

1. Design autonomy: Individual DBMSs can use the data models and transaction management techniques that they prefer.
2. Communication autonomy: Each of the individual DBMSs can make its own decision regarding the type of information it wants to provide to other DBMSs or to the software that controls its global execution.
3. Execution autonomy: Each DBMS can execute the transactions that are submitted to it in any way that it wants.

A number of alternatives are suggested below. One alternative considered is *tight-integration*, where a single-image of the entire database is available to any user who wants to share the information that may reside in multiple databases. From the user's perspective, the data is logically centralized on a database. In tightly-integrated systems, the data managers are implemented so that one of them is in control of the DBMS processing of each user request, even if a request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs, although they usually have the required functionality.

The second alternative is *semi-autonomous* systems, which consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determine what parts of their own database they will make accessible to users of other DBMSs. They are not fully autonomous systems because they must be modified to permit information exchange.

The final alternative considered is *total isolation*, where the individual systems are stand-alone DBMSs, that do not know of the existence of other DBMSs. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

It is important to note that the three alternatives considered for autonomous systems are not the only possibilities. We simply highlight the three most popular alternatives.

The distribution dimension of the taxonomy deals with data. We consider two cases: either the data is physically distributed over multiple sites that communicate with each other over a communication medium, or it is stored at only one site.

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols, to variations in data managers. The important differences considered in this thesis relate to data models, query languages and transaction management protocols. Representing data with different models creates heterogeneity because of the inherent expressive power and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (e.g., set-at-a-time access in relational systems versus record-at-a-time access in network and hierarchical systems), but also covers differences in languages even when the individual systems use the same data model. Different query languages that use the same data model often select very different methods for expressing identical requests.

The architectural alternatives are considered in turn. Starting at the origin in Figure 1.1 and moving along the autonomy dimension, the first class of systems are those which are logically integrated. Such systems can be given the generic name *composite systems* [26]. If there is no distribution or heterogeneity, then the system is a set of multiple DBMSs which are logically integrated. Shared-everything multiprocessor environments are an example of such systems. If heterogeneity is introduced, then one has multiple data managers which are heterogeneous but provide an integrated view to the user. In the past, some work was done in this class where systems were designed to provide integrated access to network, hierarchical, and relational databases residing on a single machine (see the work of Dogac and Ozkarahan [9], for example). The more interesting case is where the database is physically distributed even though a logically integrated view of the data is provided to users. This is what is known as a *distributed DBMS* [34]. A distributed DBMS can be homogeneous or heterogeneous.

Next on the autonomy dimension are semi-autonomous systems which are commonly called *federated DBMSs* [26]. The component systems in a federated environment have significant autonomy in their execution, but their participation in a federation indicate that they are willing to cooperate with others in executing user requests that access multiple databases. Similar to logically integrated systems, federated systems can be distributed or single-site, homogeneous or heterogeneous.

If one moves to full autonomy, then we get what we call *multidatabase system* (MDS) architectures. Without heterogeneity or distribution, an MDS is an interconnected collection of autonomous databases. A multidatabase management system[2]

---

[2] The term as originally suggested by Litwin [28, 29], differs from this one by assuming both heterogeneity and distribution are present in the multidatabase system.

(MDMS) is the software manages a collection of autonomous databases and provides transparent access to it. If the individual databases that make up the MDS are distributed over a number of sites, we have a *distributed MDS*. The organization and management of a distributed MDS is quite different from that of a distributed DBMS. We discuss this issue in more detail in the upcoming sections. At this point, it is sufficient to state that the fundamental difference is the level of autonomy of the local data managers. Centralized or distributed multidatabase systems can be homogeneous or heterogeneous.

The fundamental point of the foregoing discussion is that the distribution of databases, their possible heterogeneity, and their autonomy are different issues. It follows that the issues related to multidatabase systems can be investigated without reference to their distribution or heterogeneity. The additional considerations that distribution brings are no different than those of logically integrated distributed database systems for which solutions have been developed [34]. Furthermore, if the issues related to the design of a distributed multidatabase are resolved, introducing heterogeneity may not involve significant additional difficulty. This is true only from the perspective of database management; there may still be significant heterogeneity problems from the perspective of the operating system and the underlying hardware. Therefore, the more important issue is the autonomy of the databases not their heterogeneity.

The environment considered in this thesis is a multidatabase system. Thus, we assume the pessimistic case of fully autonomous DBMSs. However, we do not deal with heterogeneity and distribution issues even though they would not be difficult to add to our model.

1.2 MDS Architecture

The architectural model usually adopted to describe database management systems is the well known ANSI/SPARC framework [40] which specifies the layers of abstraction of data in a database. The ANSI/SPARC model specifies a *conceptual schema* which defines the logical organization of all the data entities in the database. The physical storage specification of these data entities are defined by the *internal schema*. The user's view of the database is specified by various *external schema* definitions. This model can be extended to distributed and multidatabase systems. The specific extensions depend upon the availability or lack of a *global conceptual schema* (GCS) that defines the data in all of the participating database. These issues are secondary to the topic of the thesis and are dealt with in Özsu and Valduriez [34] and Özsu and Barker [33]. Instead, we concentrate on the component architecture of MDSs.

The component-based architectural model of a MDMS features full-fledged DBMSs, each of which manag a different DBMS. The MDMS provides a layer of software that runs on top of these individual DBMSs and allows users to access various databases (Figure 1.2). Depending upon the existence (or lack) of the global conceptual schema or the existence (or lack) of heterogeneity, the contents of this layer of software would change significantly. Note that Figure 1.2 represents a non-distributed MDMS. If the system is distributed, one would need to replicate the multidatabase layer to each site where there is a local DBMS that participates in the system. Also note that as far as the individual DBMSs are concerned, the MDMS layer is simply another application which submits requests and receives results.

To facilitate the transaction execution model discussion in the next chapter, let us concentrate on components that are necessary for the execution of user transactions (Figure 1.3). Each of the DBMSs making up such a system has the functionality to receive user transactions, execute them to termination (either commit or abort), and report the result to the user. In this context, the term "user" refers to interactive user requests as well as application programs issuing database access commands.

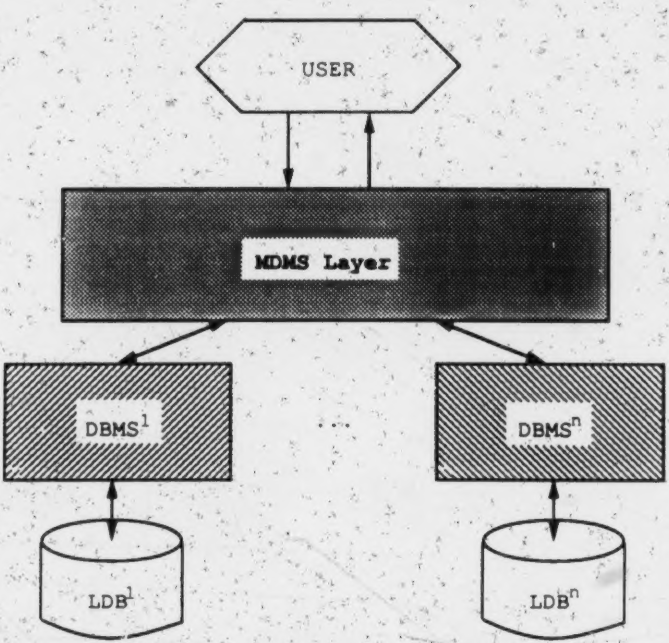


Figure 1.2: Components of an MDS

Each DBMS has its own transaction processing components. The components are a transaction manager (called the *local transaction manager* (LTM)), a *local scheduler* (LS), and a *local data manager* (LDM). This structure is quite common for centralized DBMSs (see, for example, Bernstein *et al.* [3]). The function of the LTM is to interact with the user and coordinate the atomic execution of the transactions. The local scheduler is responsible for ensuring the correct execution and interleaving of all transactions presented to the LTM. The local recovery manager ensures that the *local database* (LDB) contains all of the effects of committed transactions and none of the effects of uncommitted ones. Note that we assume each autonomous DBMS manages a single database.

From the perspective of each DBMS, the MDMS layer is simply another "user" from which transactions are received and results presented. In fact, the only type of communication among the autonomous DBMSs is via the MDMS layer. The scheduling of transactions which require multiple DBMSs is done by the MDMS layer. The

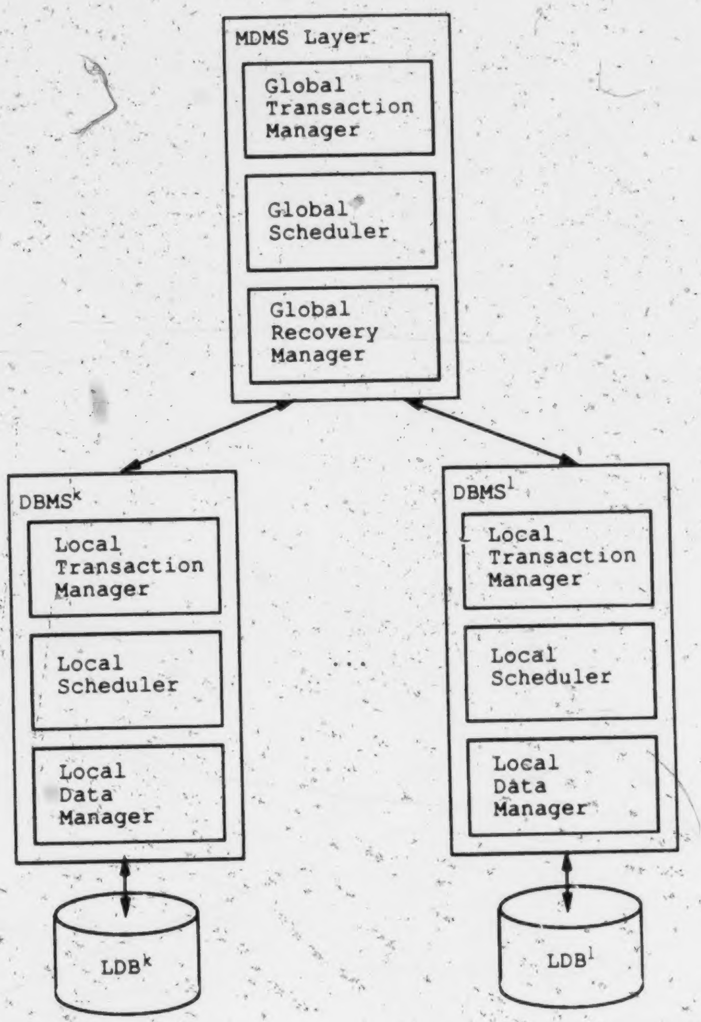

MDMS Layer
Global Transaction Manager
Global Scheduler
Global Recovery Manager
DBMS^k
Local Transaction Manager
Local Scheduler
Local Data Manager
...
DBMS^l
Local Transaction Manager
Local Scheduler
Local Data Manager
LDB^k
LDB^l

Figure 1.3: MDS Architecture

transaction manager of the MDMS layer is called the *global transaction manager* (GTM) since it manages the execution of global transactions.

1.3 Outline of Thesis

The organization of the balance of the thesis is as follows: The fundamental definitions related to transaction management and previous work specifically related to this thesis are presented in Chapter 2. Detailed study in the field requires a well defined computational model that can be used in the development of a formalism to reason about both serializability and reliability. This is provided in Chapter 3. Chapters 4 and 5 describe the issues of serialization and transactions scheduling and presents the work related to reliability. Finally, Chapter 6 makes some concluding comments and suggests directions for future research.

Chapter 2

Background

2.1 Fundamental Definitions

The notation used follows that of Özsu and Valuriez [34]. Let $O_{ij}(x)$ denote some *operation* O_j of transaction T_i operating on a database entity x. We assume that the database is static so that $O_j \in$ {read, write}. Operations are assumed to be *atomic*, in that, each is executed as an indivisible unit. We let OS_i denote the set of all operations in T_i, that is $OS_i = \bigcup_j O_{ij}$. We also denote with N_i the termination condition for T_i, where $N_i \in$ {abort, commit}[1]. A transaction now can be defined as:

Definition 2.1 *(Transaction):* A transaction T_i is a partial ordering $T_i = \{\Sigma_i, \prec_i\}$ where Σ_i is the domain consisting of the operations and the termination condition of T_i, and $\prec_i$ is an irreflexive and transitive binary relation indicating the execution order of these operations such that

1. $\Sigma_i = OS_i \cup \{N_i\}$,

---

[1] The abbreviations r, w, a and c will be used for the read, write, abort, and commit operations, respectively.

2. for any two operations $O_{ij}, O_{ik} \in OS_i$, if $O_{ij} = r(x)$ and $O_{ik} = w(x)$ for any data item x, then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$, and

3. $\forall O_{ij} \in OS_i, O_{ij} \prec_i N_i$. □

In Definition 2.1, operations O_{ij} and O_{ik}, which access a common data item, are said to be *conflicting operations*. T_i is commonly used to denote both the transaction identifier and the domain. Thus, $O \in T_i$ and $O \in \Sigma_i$ both mean "O is an operation of transaction T_i".

The data items that a transaction reads are said to constitute its *read-set* (RS). Similarly, the data items that a transaction writes are said to constitute its *write-set* (WS). Note that the read-set and the write-set of a transaction need not be mutually exclusive. Finally, the union of the read-set and the write-set of a transaction constitutes its *base-set* (BS). We adopt the notational convention of using calligraphic lettering to denote formal sets and roman fonts for acronyms. Thus, BS is an abbreviation for "base-set" whereas $\mathcal{BS}$ denotes the set of data items in the base-set of a transaction. Similarly, $\mathcal{RS}$ and $\mathcal{WS}$ can be defined for the read-set and write-set. Note that for a given transaction $\mathcal{BS} = \mathcal{RS} \cup \mathcal{WS}$.

Histories are defined with respect to transactions.

Definition 2.2 *(History):* Given a DBMS with a set of transactions $\mathcal{T}$ a history (H) is a partial order $H = (\Sigma, \prec)$ where:

(i) $\Sigma = \bigcup_j \Sigma_j$ where Σ_j is the domain of transaction $T_j \in \mathcal{T}$,

(ii) $\prec_H \supseteq \bigcup_j \prec_j$ where $\prec_j$ is the ordering relation for transaction T_j at the DBMS, and

(iii) for any two conflicting operations $p, q \in H$, either $p \prec_H q$ or $q \prec_H p$. □

Given two conflicting operations p and q, the transactions that they belong to are also said to conflict. Definition 2.2 describes the ordering of transactions on a single database management system. The definition of transactions and their history make it possible to reason about serializability and reliability on database systems. Further, it is possible to reason about serializability and reliability separately since they are related but different problems. The issues of serializability are presented in Section 2.1.1 and reliability is considered in Section 2.1.2.

2.1.1 Serializability

Database serial histories are defined as:

Definition 2.3 *(Serial):* A database history $H = \{T_1, \cdots, T_r\}$ is serial iff $(\exists p \in T_i, \exists q \in T_j \text{ such that } p \prec_H q) \models (\forall r \in T_i, \forall s \in T_j, r \prec_H s)$. □

The condition states that if an operation of a transaction precedes an operation of another transaction then all operations of the first transaction should precede any operation of the second one.

Before describing serializability it is necessary to consider when two histories are equivalent.

Definition 2.4 *(Equivalence of Histories (≡)):* Two histories are conflict equivalent[2] if they are defined over the same set of transactions with identical operations and they order conflicting operations of nonaborted transactions in the same way. □

Definition 2.5 *(Serializable):* A history (H) is serializable iff it is equivalent to a serial history. □

---

[2] Other correctness criteria have been suggested including view serializability and final-state serializability. A discussion of these is beyond the scope of the thesis but are covered in Papadimitriou [35].

Determining whether a history is serializable can best be accomplished by analyzing a graph derived from the history. "Let H be a history over $T = \{T_1, \cdots, T_n\}$. The *serialization graph* (SG) for H, denoted SG(H), is a directed graph whose nodes are the transactions in T that are committed in H and whose edges are all $T_i \rightarrow T_j$ $(i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H." [3, Page 32] The following theorem is useful when analyzing such histories.

Theorem 2.1 *(The Serializability Theorem)* A history (H) is serializable iff SG(H) is acyclic.

Proof: See Bernstein *et al.* [3, Page 33].

2.1.2 Reliability

This section describes the fundamental definitions and terminology necessary to reason about reliability. This work is based on a formalism of reliability concepts for centralized DBMSs developed by Hadzilacos [25].

It is first necessary to describe when a transaction "reads from" another transaction in an arbitrary history.

Definition 2.6 *(Reads From)* A transaction T_j reads x from T_i in a history H, if

1. $r_j(x) \in T_j, w_i(x) \in T_i$ and $w_i(x) \prec_H r_j(x)$,
2. if $a_i \in T_i \in H$ then $a_i \not\prec_H r_j(x)$, and
3. $\forall w_k(x) \in T_k$ where $w_j(x) \prec_H w_k(x) \prec_H r_i(x)$,
   $\exists a_k \in T_k$ where $w_j(x) \prec_H a_k \prec_H r_i(x)$. □

The first condition of Definition 2.6 states that the write on x by T_i precedes the read by T_j. The second condition states that transaction T_i which performs the write operation has not aborted before the read operation of T_j. Finally, the third condition guarantees that no other non-aborted transaction has updated x between the time that T_i updated the value and T_j read the value.

Hadzilacos [25] classifies histories into three categories. Each can be summarized as follows. A history or execution order is *recoverable* (RC) if every transaction that commits only reads from transactions which have already committed. A history *avoids cascading aborts* (ACA) if all transactions read from transactions which have already committed. The difference between recoverable execution orders and those that avoid cascading aborts is that, in the former, a transaction is permitted to read from an active transaction, but in the latter, a transaction must only read from committed transactions. Finally, a history is *strict* (ST) if it avoids cascading aborts and if whenever T_j writes a data item read or written by T_i, then T_j has committed or aborted.[3] Strictness differs from ACA executions in that the former only restricts read operations while the latter restricts both reads and writes.

Theorem 2.2 $ST \subset ACA \subset RC$

Proof: See Hadzilacos [25] or Bernstein *et al.* [3, Page 35]. □

Earlier we claimed that serializability was related to but different than recoverability. It is possible to create histories which are recoverable but not serializable and vice versa. The actual relationship among these histories is depicted in Figure 2.1. Examples of their inter-relationship can be found in the literature [3, 25] so they are not repeated at this time.

One final point is required for this research. "A property of histories is called *prefix commit-closed* if, whenever the property is true of history H, it is also true of history C(H'), for any prefix H' of H." [3, Page 36] The possibility of transaction

---

[3] The requirement that strict histories also avoid cascading aborts is not stated explicitly in Bernstein *et al.* [3] as it is in Hadzilacos [25], but it is clearly required. This can be seen in the proof of containment ($ST \subset ACA \subset RC$) in Bernstein *et al.* [3, Page 35].

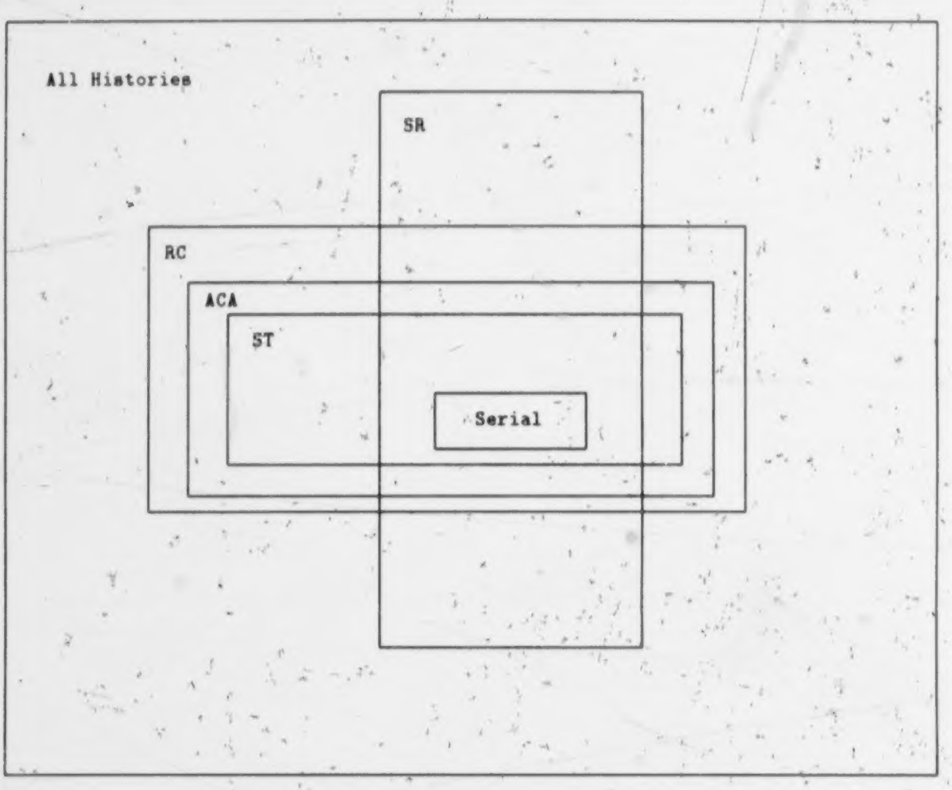

All Histories
SR
RC
ACA
ST
Serial

Figure 2.1: Relationship between histories that are SR, RC, ACA and ST

and system failures requires that the scheduler produce histories which are prefix commit-closed. If a given history (H) is correct (by some correctness criterion) but some subset of the history is not correct and the DBMS fails when only the subset has completed, the database cannot be returned to a "correct" state. It is necessary to verify that serializability, RC, ACA, and ST histories are prefix commit-closed. The schedulers developed in this thesis exhibit the prefix-commit-closed characteristic.

2.2 Summary of Related Research

Research in the multidatabase area has focused on integrating heterogeneous database systems. The majority of the research has dealt with data models and schema integration on heterogeneous databases. Research in the area of transaction management on multidatabase systems has been limited. The earliest work that addresses the issues explicitly was that of Gligor and Luckenbaugh [23] and Gligor and Popescu-Zeletin [21, 22]. In the Gligor and Popescu-Zeletin [22] paper, necessary and sufficient conditions for proper transaction management on multidatabase systems were enumerated. These conditions are self-explanatory and are stated as follows:

1. All local database management systems guarantee local synchronization atomicity.
2. Each DBMS maintains the relative order of all global transactions so that, if transaction T_1 occurs before T_2 at a DBMS, then the same is true at all others where they conflict. Recent research has demonstrated that T_1 must occur before T_2 at all DBMSs even if no conflict exists. The reasons for this will become evident later in the thesis.
3. Global transactions cannot be split and concurrently submitted to the same DBMS.

4. The MDMS must be able to identify all objects referenced by all global transactions.

5. The MDMS must be able to detect and recover from global deadlock.

All of these conditions must be met to ensure that the MDMS functions correctly.

The balance of this section describes the efforts of other researchers. Although each is identified by an individual they typically represent collaborative groups. Each researcher discussed below is recognized as a leading contributor in this specific area. Other researchers have also made contributions and will be cited where appropriate throughout the balance of the thesis.

Pu: Pu [36, 37] describes multidatabases in terms of a hierarchy of *superdatabases*. Each participating DBMS (called an *element database*) can be pictured as the leafs on a tree and each internal node is a superdatabase that manages element databases in a hierarchical structure. Each element database operates independently but global activities are managed at the node level of the tree. Transactions that cross multiple element databases are called *supertransactions* and are posed against a superdatabase.

Superdatabases utilize serializability as a transaction correctness criteria. Serializability is ensured by having each element database provide the superdatabase with information about the ordering of its local transactions. Pu claims that this is not necessary when element databases utilize strict schedulers. Serial orderings of each local transaction at the DBMS is represented by order-elements (O-elements). The order-element for a transaction is determined differently depending on what scheduler is being used at a DBMS. For example, a two phase locking algorithm assigns the order based on when the lock point[4] occurs. A timestamping scheduler would use the timestamp as the O-element.

---

[4] The lock point occurs at the end of phase one of the two-phase locking algorithm (see Eswaran *et al.* [17].

Once each O-element for every transaction, at every DBMS, is passed to the superdatabase, the ordering of supertransactions can be determined. Essentially, all of the O-elements for a supertransaction are combined into an order-vector (O-vector). Once the O-vector is formed, the ordering can be analyzed to determine if it is serializable, in which case the supertransaction can commit. Otherwise each subtransaction is aborted.

Pu's contribution is significant. Unfortunately, a number of assumptions make the work unsuitable for the environment discussed in this thesis. Formation of the O-element requires that the DBMSs inform the superdatabase of the logical time at which the locks are acquired or that the transaction is timestamped. Although this may be possible, it would require participation of the individual DBMSs, thereby violating full autonomy. The final commitment decision of supertransactions and their subtransactions is at the superdatabase level so the decision maker is not the DBMS. Crash recovery requires the direct intervention of the superdatabase and has not currently been discussed in the literature.

Summary: Pu has not yet discussed the issue of reliability. Further, Pu's research is not applicable to the environment consider in this thesis because modification of the DBMSs is necessary to provide the superdatabase with O-elements.

Breitbart and others: Breitbart *et al.* [7, 4, 5, 6] have undertaken research in database integration, query processing, architectures and transaction management on multidatabase systems. The following discussion will only consider the work as it relates to transaction management on multidatabase systems. Their work assumes data may be replicated on the multidatabase system so his approach is different from the one adopted in this research.

A global database is described as a triple $< D, S, f >$ where D is a set of global data items, S is a set of sites, and f is a function:

$$f : D \times S \rightarrow (0, 1)$$

such that $f(x, i) = 1$ means that data item x is available at site i. Further, if $f(x, i) = f(x, j) = 1$ where $i \neq j$, then x is replicated at i and j. Breitbart uses serializability as a correctness criterion and serialization graphs to determine when a history is serializable. Transactions which access multiple databases are mapped to subtransactions by a function in the following way:

1. A global read is mapped to a local read operation $r(x_i)$ for some site where $f(x, i) = 1$.
2. A global write is mapped to local write operations $w(x_{i1}), w(x_{i2}), \cdots, w(x_{il})$ where $i1, i2, \cdots, il$ are the sites that contain a copy of x.

Global operations are mapped to local operations and considered as a single history to determine if schedules are serializable. It is stated that the introduction of local transactions with these mapped global operations causes significant difficulties in scheduling.

The approach to concurrency control is to form *site graphs*. A site graph is generated for each global transaction by forming a node for each site that contains a copy of a global data item referenced by the global transaction. Once the global transaction is mapped to a set of local operations, as described above, arcs are added to the graph. If a mapped global transaction references a data item of two different sites, an arc, labeled with the global transaction identifier, is added to the global transaction's site graph. Finally, all site graphs for all global transactions are combined. If there are no cycles in the combined site graph and all local transactions only access non-replicated data, the consistency of the global data is guaranteed [6].

The most recent contribution of these researchers addresses the problems of reliability [8]. The approach is based on identifying mutually exclusive classes of globally updatable and locally updatable data. Locally updateable data is managed by the DBMS while transactions which access globally updateable data are submitted from above.

Summary: Breitbart *et al.* have proposed solutions to both the concurrency control and reliability problems. The concurrency control solution [6] is based on identifying global and local data and controlling subtransaction submission that access global data. This research takes the position that controlling submission is an appropriate technique to concurrency control but we do not differentiate between global and local data.

The reliability proposal of Brietbart *et al.* does not provide for the full autonomy of the underlying DBMSs considered by this thesis. If the DBMSs are fully autonomous they can update any data item in the their database. Therefore, the set of locally updateable data items should be the entire database. This implies that the no data item can only be globally updateable since the two sets are mutually exclusive. Thus, in a fully autonomous system, global transactions are restricted to read-only transactions.

Elmagarmid: Elmagarmid has been working in the area of transaction management on multidatabase systems since the mid-1980s. Initial work [15, 16, 14, 27] focused on characterizing the problem and proposed a number of pragmatic approaches. The research proposed maintaining a high-level of DBMS autonomy while using serializability as a correctness criterion. Autonomy is maintained by adding software to each DBMS which is responsible for ensuring that only serializable histories are produced. The monitoring and submitting software was known as a STUB process.

Elmagarmid proposed WEAK-STUB/STRONG-STUB processes for ensuring correct execution of global transactions [15]. Essentially the approach was to submit a subtransaction as a WEAK-STUB process if it could be aborted by the DBMS or as a STRONG-STUB process once the subtransaction had to commit. Unfortunately, scenarios can be envisioned which would require commitment at the global level, but an autonomous DBMS could continuously refuse to commit the transaction.

Du and Elmagarmid [10] recognized the difficulties of using serializability as a correctness criterion. The basic problem is that two global transactions which do not reference common data items can conflict. Such conflicts are known as *indirect conflicts* and cannot be detected at the MDMS level. The recognition of indirect conflicts led to a correctness criteria which they called *quasi-serializability* (QSR) [11, 10]. Because of its similarity to the work reported in this thesis, quasi-serializability will be discussed in greater detail later in this thesis (Section 4.3).

Du and Elmagarmid subsequently merged the pragmatic approach of this early work with the new correctness criterion [10, 13]. The approach suggested is to have input both the transaction and a prespecified order and produce an execution order. The STUB process then takes the global transactions and the execution order and submits the portion pertinent to the DBMS as a global subtransaction. In this way, the STUB process can determine an execution order for the DBMS without having to modify the underlying DBMS. The research suggests how this would work with a number of different schedulers. Improvements in processing time can be made if the local scheduler is modifiable and techniques for this are also discussed in their papers.

Summary: Elmagarmid's research is the most closely related work to this thesis. This work has approached the problem in a different way and has produced schedulers and a detailed description of reliability. The thesis refers to their work and presents it in more detail where appropriate. Elmagarmid's work does not consider reliability concerns to the same extent as this thesis.

Rusinkiewicz and others: Rusinkiewicz has recently made a contribution in the area of reliable multidatabase transactions [19, 20]. The work uses serializability as a correctness criterion for concurrency control and attempts to solve the problem of indirect conflicts. A class of histories which is more restrictive than strict, called *rigorous* is introduced. Rigorous histories are those where the serialization order of the transactions is the same as their execution order. Their solution is to force local schedulers to produce rigorous schedules.

Summary: The work is very new and significant because it uses serializability as a correctness criterion and may solve the problem of indirect conflicts. Rigorousness may be too restrictive, but the research possibilities are interesting. The work is also important in that it addresses the atomicity of multidatabase transactions.

Other Work: There are a number of other transaction management proposals that are relevant to the topic of this thesis. Some of these proposals deal directly with transaction management on multidatabase systems, while others address issues related to long lived transactions but may be applicable to the environment that we are considering. A brief review of these proposals are provided in this section.

Litwin and Tirri [30] propose the use of timestamps to determine when transactions execute concurrently. A data item is assigned a *value date* which indicates the time that the item was given a correct value. When transactions are created an *actual date* is assigned to them and if the transactions actual data and data items value dates indicate safe access is possible the transaction is permitted to execute. The approach is able to delegate the synchronization of global transactions to the local transaction managers. However, it assumes that the individual DBMSs timestamp data items and are able to compare transaction timestamps with data timestamps. Since this thesis assumes full autonomy of the underlying DBMSs the research proposed by Litwin and Tirri is not applicable.

Nested transactions [32] has been suggested as an approach to transaction management in the multidatabase environment. A full discussion of nested transactions is beyond the scope of this thesis but a brief summary of the approach is presented. Transaction which span multiple databases are split into subtransactions and submitted to their respective DBMSs. Execution of each subtransaction is then the responsibility of each DBMS. However, the entire nested transaction is executed as an atomic unit and its results are not available to the outside world until all the subtransactions commit. Nested transaction managers must have control of all transaction and subtransactions to function correctly. The transaction model that is defined in this thesis can actually be seen as a two-level nested transaction model. However, multidatabase systems are more complex since local transactions are submitted independently.

An alternative nested transaction model intended to deal with long lived transactions has been proposed by Garcia-Molina and Salem [18]. Their model uses nested transactions called *sagas*, where with only two levels of nesting. In this sense, a saga is similar to the transaction model proposed in this thesis. Further sagas differ from Moss's nested transaction model in that a saga is "open" since it is not executed as an atomic unit. Therefore, the effects of a subtransaction's commitment are visible as soon as without having to wait until the commitment of the entire saga. It is also possible that a saga's transaction may need to be rolled-back to abort the saga and this is accomplished by means of compensating transactions. This means that sagas are written so they can interleave with any others, which makes concurrency control at the saga level unnecessary. Since sagas can interleave with any other transaction, the introduction of local transactions to the model does not pose additional difficulty. Inherent in the model are two assumptions which make the approach undesirable for the multidatabase environment. First, the model is not applicable to all multidatabase environments since it may be too restrictive to require that sagas be interleavable with any other transaction. Secondly, it may not always be possible to write compensating transactions. For example, a real time application which drills a hole or fires a missile is probably not compensatable.

Another approach to long lived transactions is *altruistic locking* [1, 18] which similar to our scheduling algorithms so the discussion is deferred until Chapter 4.

Chapter 3

A Formal Model of MDS

This chapter provides the fundamental definitions required to characterize multidatabase systems. Whenever possible, existing definitions from traditional DBMS research have been extended to the MDS environment. The chapter is presented by first describing the types of transactions found on a MDS. Second, the histories on a MDS are formally defined (Section 3.2). Once these basic definitions are presented, the definitions required to reason about serializability are provided (Section 3.3). The formalization necessary to ensure that MDSs are reliable is then introduced (Section 3.4). Finally, the computational model is summarized by stating the assumptions required by the research. The contents of Section 3.1-3.4 have recently been published [33].

3.1 Transaction Types

A multidatabase system contains two types of transactions: *local* ones that are submitted to each DBMS and *global* ones that are submitted to the MDMS. Local transactions execute on a single database while global ones access multiple databases or non-local data. Global and local transactions are formally defined by the composition of their base-sets and their mode of operation. This requires the following definition.

Definition 3.1 *(Local Database):* Each of the autonomous databases that make up a multidatabase is called a *local database* (DBMS). The set of data items stored at a DBMS, say i, is denoted $\mathcal{LDB}^i$. The set of all data in the multidatabase can be defined as:

$$\mathcal{MDB} = \bigcup_i \mathcal{LDB}^i. \quad \square$$

Local transactions only access data at the DBMS where they are posed. They are always submitted to and executed by one DBMS. Global transactions are those that require access to more than one database or to a database which is not controlled by the one where they are posed.

Definition 3.2 *(Local transaction):* A transaction T_i submitted to DBMS j (denoted $DBMS^j$) is a *local transaction* (denoted LT_i^j) on $DBMS^j$ if $\mathcal{BS}_i \subseteq \mathcal{LDB}^j$. $\square$

We denote the set of all local transactions at $DBMS^j$ by $\mathcal{LT}^j = \bigcup_i LT_i^j$. The set of all local transactions in a multidatabase system is $\mathcal{LT} = \bigcup_j \mathcal{LT}^j$.

Definition 3.3 *(Global transaction):* A transaction is a *global transaction* (GT_i) iff:

(i) $\nexists LDB^j$ such that $\mathcal{BS}_i \subseteq \mathcal{LDB}^j$ or

(ii) GT_i is submitted to $DBMS^k$ but $\mathcal{BS}_i \subseteq \mathcal{LDB}^r$ $(k \neq r)$. $\square$

Item (i) states that global transactions, submitted to the MDMS, access data items stored in more than one database. Item (ii) represents the case where a user working on one DBMS requires access to the data stored and managed by another. We let $\mathcal{GT}$ be the set of all global transactions, i.e. $\mathcal{GT} = \bigcup_i GT_i$.

Global transactions are parsed into a set of *global subtransactions* which are submitted to the local DBMSs for execution. Thus, a global transaction is executed as a set of subtransactions at a number of local DBMSs. We define global subtransactions in terms of the data items referenced (i.e. their base-set) and the global transaction creating them.

Definition 3.4 *(Global Subtransaction)* A global subtransaction submitted to $DBMS^j$ for global transaction GT_i (denoted GST_i^j) is a transaction where:

(i) $\Sigma_i^j \subseteq \Sigma_i$ and

(ii) $\mathcal{BS}_i^j \subseteq \mathcal{LDB}^j$, where $\mathcal{BS}_i^j$ is the base-set of GST_i^j. □

Definition 3.4 implies that each global subtransaction executes at only one DBMS. Therefore, a global subtransaction can be seen as a local transaction by the DBMS where it is submitted. We make the assumption that a global transaction only submits one subtransaction to any single DBMS.

The set of all global subtransactions produced by a global transaction, say GT_i, is denoted by $\mathcal{GST}_i$. The set of all global subtransactions submitted to a particular database, say $DBMS^k$, is denoted $\mathcal{GST}^k$. Therefore, the set of all global subtransactions in a multidatabase system is $\mathcal{GST} = \bigcup_i \mathcal{GST}_i = \bigcup_k \mathcal{GST}^k$.

Recall from Section 2.1.2 that transactions *conflict* if they access the same data item and at least one of them is a write operation. Two global transactions can conflict if their base-sets intersect or if there exists a sequence local transactions which conflict with both global transactions. Du and Elmagarmid [10] call the former *direct* and the latter *indirect* conflict.

The computational model is summarized as follows. A global transaction is managed by the MDMS layer which parses it into a set of global subtransactions. Each subtransaction is submitted to a local DBMS. Local DBMSs are responsible for

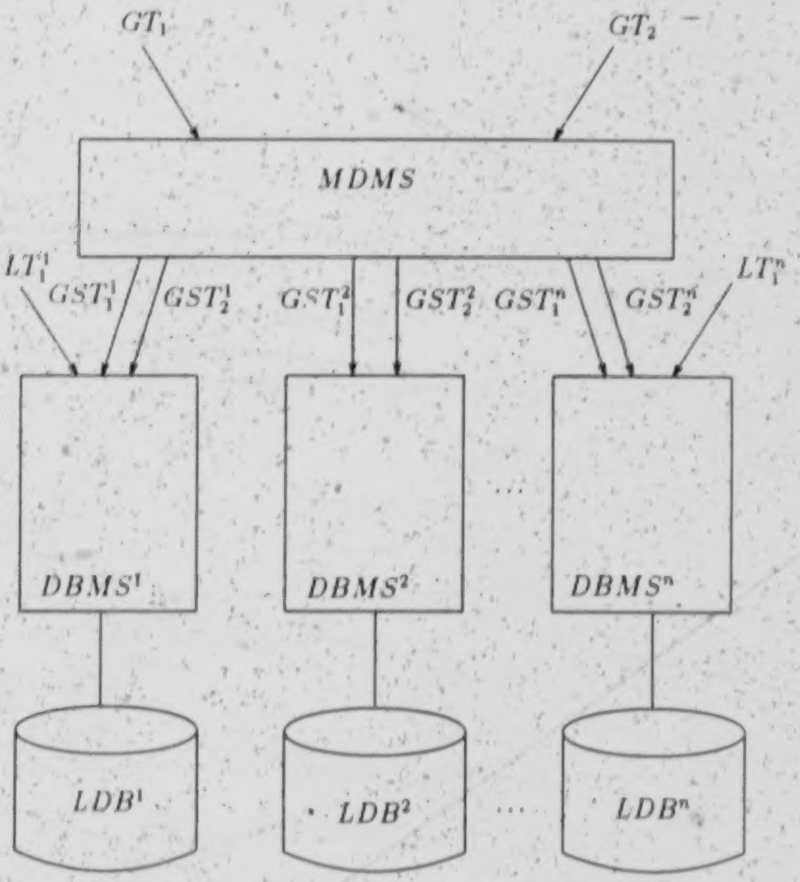

GT_1
GT_2
$MDMS$
LT_1^1
GST_1^1
GST_2^1
GST_1^2
GST_2^2
GST_1^n
GST_2^n
LT_1^n
$DBMS^1$
$DBMS^2$
…
$DBMS^n$
LDB^1
LDB^2
…
LDB^n

Figure 3.1: Depiction of the Computational Model

the concurrent execution of both the global subtransactions and the local transactions submitted to them. The synchronization of global transactions is the responsibility of the MDMS layer. The computational model is depicted in Figure 3.1.

3.2 Histories

Before discussing the difficulties of serialization and reliability in a multidatabase environment, it is necessary to define a *multidatabase history* (MH). A multidatabase history is defined in terms of "local" histories, at each DBMS, and the execution history of global transactions.

Two types of transactions exist in a multidatabase system: local and global. The histories are defined with respect to these types of transactions.

Definition 3.5 *(Local History):* Given a $DBMS^k$ with a set of local transactions $\mathcal{LT}^k$ and a set of global subtransactions $\mathcal{GST}^k$, a *local history* (LH^k) is a partial order $LH^k = (\Sigma^k, \prec^k_{LH})$ where:

(i) $\Sigma^k = \bigcup_j \Sigma^k_j$ where Σ^k_j is the domain of transaction $T_j \in \mathcal{LT}^k \bigcup \mathcal{GST}^k$ at $DBMS^k$.

(ii) $\prec^k_{LH} \supseteq \bigcup_j \prec^k_j$ where $\prec^k_j$ is the ordering relation for transaction T_j at $DBMS^k$, and

(iii) for any two conflicting operations $p, q \in LH^k$, either $p \prec^k_{LH} q$ or $q \prec^k_{LH} p$. □

Definition 3.5 describes the ordering of transactions on a MDB, but two levels of transactions are being captured. The collection of global subtransactions at each participating local database is sufficient to fully describe the ordering of global transactions. The global subtransaction history is defined as a subset of the local site histories by restricting Definition 3.5.

Definition 3.6 *(Global Subtransaction History)* The global subtransaction history of a $DBMS$ say k, is defined by the partial order $GSH^k = (\Sigma^k_{GSH}, \prec^k_{GSH})$ where:

(i) $\Sigma^k_{GSH} = \bigcup_j \Sigma^k_j$, where Σ^k_j is the domain of transaction $T_i \in \mathcal{GST}^k$ and

(ii) $\prec^k_{GSH} \subseteq \prec^k_{LH}$. □

A global transaction history can be defined based on the ordering of global subtransactions at each local database. A global history is the union of the global subtransaction histories at each participating local database as illustrated by the following definition:

Definition 3.7 *(Global History)* A global history $GH = (\Sigma_{GH}, \prec_{GH})$ is the union of all global subtransaction histories:

(i) $\Sigma_{GH} = \bigcup_k \Sigma^k_{GSH}$,

(ii) $\prec_{GH} \supseteq \bigcup_k \prec^k_{GSH}$, and

(iii) for any two conflicting operations $p, q \in GH$, either $p \prec_{GH} q$ or $q \prec_{GH} p$. □

The final definition describes the history of a multidatabase execution. Essentially the MDB history is fully described by the combination of local site histories (Definition 3.5) and the global history (Definition 3.7).

Definition 3.8 *(MDB History):* A history of a multidatabase (denoted MH) consisting of n local histories and a global history (GH) can be described as a tuple $MH = < \mathcal{LH}, GH >$ where $\mathcal{LH} = \{LH^1, LH^2, \cdots, LH^n\}$. □

The following example illustrates the application of these definitions to the MDB environment. Whenever possible this example will be used to illustrate serializability and reliability concepts throughout the thesis.

Example 3.1 Assume that a multidatabase system is composed of two local databases whose contents are: $\mathcal{LDB}^1 = \{d, e, f, g\}$ and $\mathcal{LDB}^2 = \{s, t, u, v\}$. Two global transactions are posed to the multidatabase as follows:

$$GT_1 : r_1(d); r_1(e); w_1(s); w_1(d); c_1$$
$$GT_2 : r_2(d); r_2(u); w_2(s); w_2(d); c_2$$

These generate the following global subtransactions:

$$GST_1^1 : r_1^1(d); r_1^1(e); w_1^1(d); c_1^1$$
$$GST_1^2 : w_1^2(s); c_1^2$$
$$GST_2^1 : r_2^1(d); w_2^1(d); c_2^1$$
$$GST_2^2 : r_2^2(u); w_2^2(s); c_2^2$$

Further, we introduce local transactions into each DBMS as follows:

$$LT_1^1 : \hat{r}_1^1(e); \hat{w}_1^1(e); \hat{w}_1^1(d); \hat{c}_1^1;$$
$$LT_1^2 : \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2;$$

The $\hat{r}$ ($\hat{w}$) notation distinguishes local transactions from global subtransactions in this discussion. There is no difference between an operation of a local transaction and a global subtransaction; the differentiation is for notational convenience.

Assume the following local histories are produced by the local schedulers at each DBMS.

$$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); \hat{c}_1^1; c_2^1; c_1^1;$$
$$LH^2 : r_2^2(u); w_2^2(s); w_1^2(s); \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2; c_1^2; c_2^2;$$

The following global subtransaction histories can be derived from these local histories:

$$GSH^1 : r_1^1(d); r_1^1(e); w_1^1(d); r_2^1(d); w_2^1(d); c_1^1; c_2^1;$$
$$GSH^2 : r_2^2(s); w_2^2(s); w_1^2(s); c_1^2; c_2^2;$$

Finally, the global history is the partial order which combines GSH^1 and GSH^2 as $GH = \{GSH^1 \cup GSH^2\}$. The multidatabase history is the tuple $MH = < \{LH^1, LH^2\}, GH >$. ◇

3.3 MDB-Serializability

Multidatabase systems are composed of local histories which collectively contain a global history. Serializability, as a correctness criteria, is not appropriate for the MDB environment because it is intended to model transactions contained in a single history. By definition, multidatabase systems are composed of multiple histories, so a new correctness criterion is proposed. The proposed correctness criterion captures both the local histories and the history of transactions which are not completely contained at a single DBMS. The following definitions present the correctness criterion.

Definition 3.9 *(MDB-Serial):* A multidatabase history is MDB-Serial iff:

(i) every $LH \in \mathcal{LH}$ is (conflict) serializable, and

(ii) given a $GH = \{GST_1^n, \cdots, GST_r^m\}$, if $\exists p \in GST_i^k, \exists q \in GST_j^k$ such that $p \prec_{GH} q$, then $\forall k, \forall r \in GST_i^k, \forall s \in GST_j^k, r \prec_{GH} s$. □

The first condition states that local histories are conflict serializable. It is not necessary to require that local histories be serial since we assume each local transaction manager can serialize submitted transactions. The second condition states that if an operation of a global transaction precedes an operation of another global transaction in one local history, then all operations of the first global transaction must precede any operation of the second in all local histories.

Definition 3.10 *(Equivalence of Histories ($\equiv$)):* Two histories are conflict equivalent if they are defined over the same set of transactions and they order conflicting operations of nonaborted transactions in the same way. □

Definition 3.11 *(MDB-Serializable (MDBSR)):* A MH is MDB-serializable iff it is equivalent to a MDB-Serial history. □

3.4 MDS Recoverability

This section describes the fundamental definitions and terminology necessary to reason about multidatabase recovery. The work is based on a formalism of reliability concepts for centralized DBMSs [25]. Local and global level recoverability are described in Section 3.4.1 and Section 3.4.2, respectively.

3.4.1 Local Level Recoverability

Example 3.2 Recall Example 3.1. Different levels of recoverability at the local DBMSs are illustrated by a sequence of increasingly restrictive histories. Assume the following local history is produced $DBMS^1$'s scheduler.

$$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); \hat{c}_1^1; c_2^1; c_1^1; \qquad (H_1)\diamond$$

Definition 3.12 The execution sequence represented by a local history is *local recoverable* (LRC) if, every transaction that commits reads only from a transaction which has already committed.□

Example 3.3 H_1 is not recoverable because c_2^1 precedes c_1^1. This can be corrected as follows:

$$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); \hat{c}_1^1; c_1^1; c_2^1; \qquad (H_2)\diamond$$

Although the execution above is locally recoverable it does not avoid cascading aborts.

Definition 3.13 A local history *avoids local cascading aborts* (ALCA) if, all transactions read from transactions which have committed. □

Example 3.4 The history H_2 is not ALCA because $w_1^1(d) \prec r_2^1(d) \prec c_1^1$ so GST_2^1 reads from GST_1^1 before GST_1^1 commits. The following history is ALCA:

$$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); c_1^1; r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); \hat{c}_1^1; c_2^1; \qquad (H_3) \ \diamond$$

The most restrictive histories are locally strict.

Definition 3.14 A history is *locally strict* (LST) if, it avoids cascading aborts and if whenever $w_j(x) \prec o_i(x) (i \neq j)$ then $n_j \prec o_i(x)$, where $n_j \in \{o_j, c_j\}$ and $o_i(x)$ is $r_i(x)$ or $w_i(x)$. □

Example 3.5 H_3 is not locally strict since $\hat{w}_1^1(d) \prec w_1^1(d) \prec \hat{c}_1^1$. The local history:

$$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); c_1^1; r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); \hat{c}_1^1; w_2^1(d); c_2^1; \qquad (H_4)$$

is LST since LT_1^1 commits before GST_2^1 updates the value of d. ◇

LRC, ALCA, and LST are a direct extension of the definitions given by Hadzilacos [25]. Therefore, the relationship $LST \subset ALCA \subset LRC$, proven by Hadzilacos, is valid in this environment (Figure 3.2). Global recoverability is not as straightforward.

3.4.2 Global Recoverability

Global recoverability requires two conditions be met:

1. All local histories are local recoverable.
2. All global subtransactions submitted on behalf of a global transaction have the same termination condition.

The second condition is summarized by the following definition.

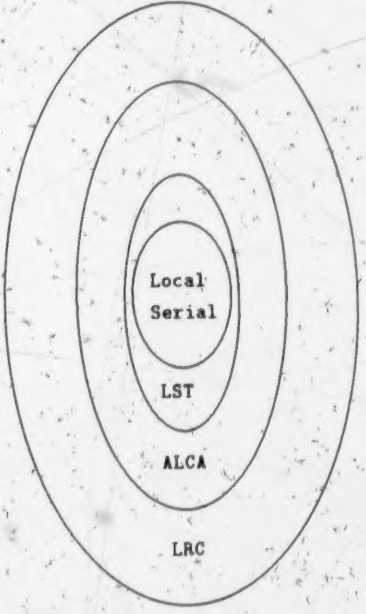


Figure 3.2: Recoverability Containment Properities of Local Histories

Definition 3.15 *(Global Transaction Termination Uniformity)* A global transaction (GT_i) terminates uniformly if either one of the following conditions hold:

1. if $\exists GST_i^l \in \mathcal{G}ST^l$ where $a_i^l \in GSH^l$, then
   $\forall GST_i^k$ where the write set of GST_i^k is not empty, $\nexists c_i^k \in GSH^k$.
2. if $\exists GST_i^l \in \mathcal{G}ST^l$ where $c_i^l \in GSH^l$, then
   $\forall GST_i^k$ where write set of GST_i^k is not empty, $\nexists a_i^k \in GSH^k$. $\Box$

Definition 3.15 part (1) means that if any global subtransaction has aborted at any DBMS, then all other global subtransactions belonging to that global transaction have either aborted or they have not yet terminated. Part (2) means that if any global subtransaction has committed at any DBMS, then all other global subtransactions have either committed or are still active. The restriction to the write set in the above definition is required only because it simplifies the discussion and later implementation. This is because read-only transactions will not affect the local databases as they can be ignored when considering reliability.

Definition 3.16 *(Global Recoverability)* A global history is *globally recoverable* (GRC) if,

1. all LH^i are at least LRC and and
2. all global transactions terminate uniformly. □

Condition (1) requires local histories to be at least locally recoverable. For the purpose of GRC, we are interested in the set of GSHs which comprise the global history. However, if an arbitrary local history LH^i is locally recoverable it follows that any subset of LH^i exhibits the same property. Although this may be more restrictive than necessary, it is the only way to guarantee that global subtransaction histories are recoverable given the autonomy of the DBMSs. Since $GSH^i \subseteq LH^i$ it follows that GSH^i is also LRC. This implies that a more restrictive history can occur in a GRC history. For example, a MDMS which guarantees LRC at $DBMS^1$, ALCA at $DBMS^2$ and LST at $DBMS^3$ could provide GRC histories (see Figure 3.3 (a)). Condition (2) provides consistency across DBMS boundaries.

Since avoidance of cascading aborts and strictness are subsets of recoverable histories we have the following definitions.

Definition 3.17 *(Avoids Global Cascading Aborts)* A global history *avoids global cascading aborts* ($AGCA$) if,

1. all LH^i are at least $ALCA$ and
2. all global transactions terminate uniformly. □

The types of local histories permitted in a AGCA history are depicted in Figure 3.3(b).

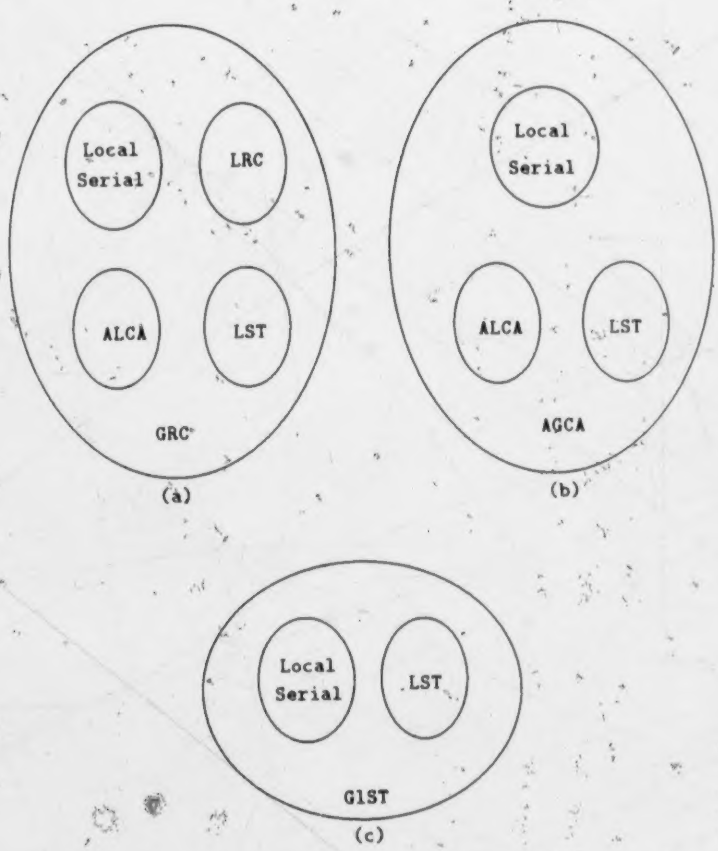

Local
Serial
LRC
ALCA
LST
GRC
(a)
Local
Serial
ALCA
LST
AGCA
(b)
Local
Serial
LST
GlST
(c)

Figure 3.3: Types of Local Histories Possible in a Global History which are (a) GRC, (b) AGCA, and (c) GlST

Definition 3.18 *(Globally Strict)* A global history is *globally strict* ($GlST$) if,

1. all LH^i are at least LST and
2. all global transactions terminate uniformly. □

The types of local histories permitted locally in a GlST history are depicted in Figure 3.3 (c).

The major difference between global recoverable histories and local recoverable histories is that all global subtransactions, for any global transaction, terminate in the same way. Consider the following histories:

$$LH^1 : r_1^1(d); r_1^1(e); w_1^1(d); \hat{r}_1^1(e); r_2^1(d); \hat{w}_1^1(e); \hat{w}_1^1(d); w_2^1(d); \hat{c}_1^1; c_2^1; c_1^1;$$
$$LH^2 : r_2^2(u); w_2^2(s); w_1^2(s); \hat{r}_1^2(u); \hat{w}_1^2(u); \hat{c}_1^2; c_1^2; a_2^2;$$

Both LH^1 and LH^2 are LRC but the global history is not GRC. Consider the following projection of these histories:

$$GSH^1 : r_1^1(d); r_1^1(e); w_1^1(d); r_2^1(d); w_2^1(d); c_2^1; c_1^1;$$
$$GSH^2 : r_2^2(u); w_2^2(s); w_1^2(s); c_1^2; a_2^2;$$

Since GST_2^1 commits in GSH^1 but GST_2^2 aborts in GSH^2 it is evident that $GH = GSH^1 \bigcup GSH^2$ is not GRC. If LH^2 was executed as follows:

$$GSH^2 : r_2^2(u); w_2^2(s); w_1^2(s); c_1^2; c_2^2;$$

the GH would be GRC. Similar arguments could be made for AGCA and GlST.

Hadzilacos [25] correctly argues that $LST \subset ALCA \subset LRC$ as depicted in Figure 3.2. Precisely the same argument can be made for global level recoverable histories.

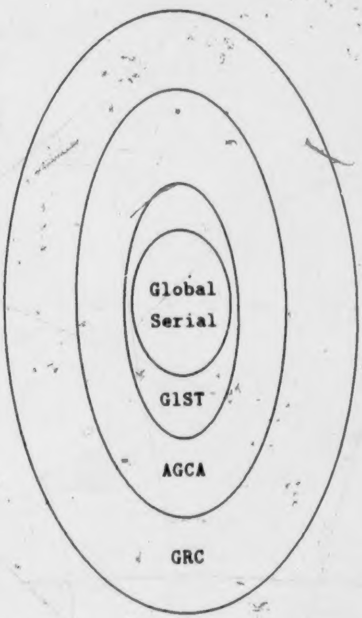


Figure 3.4: Recoverability Containment Properties of Global Histories

Theorem 3.1 GST ⊂ AGCA

Proof: All local histories in a GST history must be either local serial or local strict (Definition 3.18) and histories which avoid global cascading aborts permit ALCA local histories (by Definition 3.17). It follows that $GST \subset AGCA$. □

Theorem 3.2 AGCA ⊂ GRC

Proof: All local histories in a AGCA history must be either local serial, local strict, or avoid local cascading aborts (Definition 3.17) and histories which are GRC permit LRC local histories (by Definition 3.16). It follows that $AGCA \subset GRC$. □

From Theorem 3.1 and Theorem 3.2 we have the relationship GST ⊂ AGCA ⊂ GRC depicted in Figure 3.4.

A final comment about recoverability is required. Unlike serializability it is not meaningful to discuss recoverability in terms of "MDB-recoverability". This is a consequence of the possibility that a MDS could provide LST at one DBMS, ALCA at another while only guaranteeing GRC at the MDMS. We will discuss recoverability in terms of the level at which the transactions were submitted. That is, local versus global level recoverability, where the former refers to a local history and the associated transactions managed by a DBMS, while the latter refers to a global history and the global transactions managed by the MDMS.

3.5 Summary of Assumptions

Implicit in this architecture and computational model are certain assumptions about the system architecture and how users interact with the MDS. The following is a summary of these assumptions.

1. *Local Autonomy.* The individual DBMSs are assumed to be fully autonomous. This implies that they cannot be modified in any way nor can they communicate with each other directly. Autonomy also requires that each transaction execute until termination. In the event of any failure, each DBMS is able to fully recover autonomously and correctly without user input.
2. *Heterogeneity.* In this study and model, no assumptions are made about heterogeneity. For example, the data models, user interfaces, and transaction management policies of each of these DBMSs may be different. The central aspect of this research is autonomy, which subsumes heterogeneity.
3. *Subtransaction Decomposition.* The model assumes that a number of subtransactions execute on various databases on behalf of a global transaction. This raises the question about the feasibility of such a decomposition. Although this is a legitimate and interesting research question we do not address it in this research. Our position is that either such decomposition algorithms are available, or the global transaction model readily lends itself to the identification of

individual subtransactions (e.g. the nested transaction model [32]).

4. *Data Replication.* Data replication across multiple databases is not considered in this model. This is a corollary of the way global subtransactions are defined (i.e. they execute at only one site).

5. *Value Dependence.* It is assumed that there are no value dependencies between data items stored in different databases. In other words, integrity and consistency constraints are defined locally on each database. This is a direct consequence of the requirement for complete local autonomy of individual DBMSs. A GST can still fail however, as a result of a local value dependency. Certain recovery techniques have been developed which assume implicitly that the MDMS knows all value dependencies at each local DBMS. This research does not make this assumption, since it reduces full autonomy, but one of the approaches discussed later requires that all value dependencies on globally accessible data be known by the MDMS.

6. *Multiple Subtransactions.* A global transaction can not submit multiple global subtransactions to a single DBMS. This restriction should not be too confining since the global subtransactions can be created so that only one subtransaction is required per local database.

7. *System Failures.* A failure of the underlying operating system can cause the failure of a single DBMS while not effecting another. This means that, for example, the MDMS might fail but all DBMSs continue to operate normally. When a system failure occurs, it is the responsibility of the system to notify the DBMS administrator when the DBMS can become operational.

8. *Catastrophic or Media Failures.* When a local DBMS suffers loss of data stored on secondary storage, it does not require the assistance of the MDMS to restore the local database to a consistent state. This is a direct consequence of local autonomy and each DBMS's view that the MDMS is a user. Techniques for

recovering from media failures are known but are beyond the scope of this research.

Each assumption is necessary to limit scope of the research. Unfortunately, each restricts the environment under consideration in some way. Assumptions 1 and 2 are mandatory because they describe explicitly the type of systems being studied. Assumption 3 is an open research problem but the techniques for decomposing global transactions into subtransactions are being studied. Data replication (Assumption 4) is an immediate consequence of the MDB environment. This thesis argues that understanding a non-replicated multidatabase system is necessary before investigating a replicated one. Value dependencies which span DBMS boundaries should not occur in an MDB environment. Multidatabase systems are intended to allow sharing across autonomous DBMSs which are not aware of each other's existences so any value dependencies crossing boundaries are imposed by the MDMS and are therefore avoidable. If value dependencies were permitted to span boundaries, additional assumptions about the underlying DBMSs would be necessary which violate the full autonomy assumption. Assumption 6 is necessary to ensure that all operations of a global transaction are executed in the order defined by the transaction. Finally, Assumptions 7 and 8 are a direct consequence of Assumption 1.

Chapter 4

Multidatabase Concurrency Control

Centralized or distributed database serializability assumes the existence of a single scheduler, or a protocol among schedulers, which guarantees database consistency. As a result, traditional conflict serializability is typically utilized as a correctness criterion. This may be inappropriate in the MDB environment since each local scheduler is not capable of ensuring the consistency of global transactions. Two levels of serializability are necessary: one for each local history and one for the global history. The following example demonstrates that an MDB history is more than the collection of its local histories.

Example 4.1 Recall Example 3.1. Notice that both of the local histories are serializable. The multidatabase history is not serializable however, since the global transaction order at each DBMS is inconsistent. This can be detected using the GH portion of the multidatabase history tuple:

$$\mathrm{GH} = \quad \{\{r_1^1(d); w_1^1(d); r_2^1(d); w_2^1(d); c_1^1; c_2^1\} \cup \{w_2^2(s); w_1^2(s); c_2^2; c_1^2\}\}$$

specifically the relations of the operations

$$r_1^1(d) \prec_{GH} w_2^1(d) \text{ at } DBMS^1 \text{ and}$$
$$w_2^2(s) \prec_{GH} w_1^2(s) \text{ at } DBMS^2$$

which implies

$$GST_1^1 \prec_{GH} GST_2^1 \text{ and}$$
$$GST_2^2 \prec_{GH} GST_1^2,$$

respectively. These two serialization orders are contradictory; $DBMS^1$ specifies that global transaction GT_1 precedes GT_2 while $DBMS^2$ specifies the reverse. Thus, although the local schedules are serializable, the execution order described in GH is not, so the MDB history is not MDB-serializable. ◇

This example illustrates the difficulty of ensuring correct serialization when only GTs are present. Du and Elmagarmid [10] have demonstrated that when local transactions are introduced, the creation of a scheduler which produces serializable schedules on a MDS is unlikely (See Section 4.3). The problem results from indirect conflicts and an example appears later in this chapter.

The issue of reliability in the multidatabase environment has not been studied in detail. The only two studies that address the issue are by Breitbart *et al.* [8] and Georgakopoulos and Rusinkiewicz [20]. The balance of this chapter provides a new formalism (Section 4.1) and illustrates its correctness (Section 4.2). The correctness criterion is then compared to another similar approach (Section 4.3). Two new scheduling algorithms are presented which maintain the autonomy of the DBMSs and are demonstrated to be correct (Section 4.4). The contents of this chapter will soon be published [2].

4.1 Multidatabase Serializability Graphs

To reason about the MDB-serializability of multidatabase histories, we use a variation of the serializability graph adapted to the computational model described previously. A multidatabase history (MH) is composed of two types of transactions: namely, local transactions and global subtransactions. Therefore, both types of transaction histories must be modeled: the set of local histories ($\mathcal{LH}$) and the global history (GH) The MDB-serializability of a multidatabase history can be represented by a directed graph defined by $G = (V_1, V_2, A_1, A_2)$, where V_1 is a set of vertices, one for each global subtransaction in MH and V_2 is a set of vertices, one for each local transaction in MH. A_1 is a set of arcs representing the ordering of transactions in local histories. Thus, these arcs connect vertices in $V_1 \bigcup V_2$. A_2 is a set of arcs connecting vertices in V_1 according to the execution order in the global history GH. The arcs are formed such that, for any two conflicting transactions T_i and T_j in a history where $T_i \prec T_j$, an arc $T_i \rightarrow T_j$ is formed. Note that if two GSTs conflict, an arc exists in both A_1 and A_2.

Example 4.2 Recall Example 3.1 and 4.1. Four global subtransactions are present in the histories, two in each local history. Two local transactions are also present, one in each local history. The graph defined above, $G = (V_1, V_2, A_1, A_2)$, as applied to Example 3.1 is:

$$V_1 = \{GST_1^1, GST_2^1, GST_1^2, GST_2^2\}$$
$$V_2 = \{LT_1^1, LT_1^2\}$$

From the set of local histories ($\mathcal{LH}$) the arcs of A_1 are:

$$\begin{aligned} A_1 = \quad & \{GST_1^1 \rightarrow LT_1^1, \\ & GST_1^1 \rightarrow GST_2^1, \\ & GST_2^2 \rightarrow GST_1^2, \end{aligned}$$

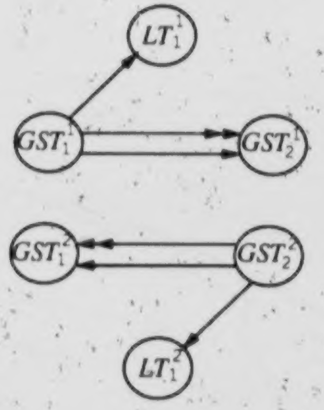


Figure 4.1: Intermediate Graph Representation

$$GST_2^2 \rightarrow LT_1^2\}$$

Finally, from the global history (GH) the arcs of A_2 are:

$$\begin{aligned} A_2 = \quad & \{GST_1^1 \rightarrow GST_2^1, \\ & GST_2^2 \rightarrow GST_1^2\} \end{aligned}$$

Figure 4.1 depicts the graph using single headed arrows for A_1 arcs and double headed arrows for A_2 arcs.◇

This representation can be simplified to characterize multidatabase histories. It is possible to represent all of the GSTs of a particular GT as a single vertex while retaining the vertices representing LTs. The arcs are modified accordingly. The remainder of the thesis uses these simplified ***multidatabase serializability graphs*** (MSG) to reason about multidatabase histories. The following definition describes the derivation of these graphs for multidatabase histories.

Definition 4.1 *(Multidatabase Serializability Graph)* Given an arbitrary multidatabase history (MH), its multidatabase serializability graph is a digraph defined with the ordered four: $MSG(MH) = (\Gamma, \Lambda, \gamma, \lambda)$. Each element of the ordered four is defined as follows:

1. Γ is a set of labeled vertices representing global transactions.[1]

2. Λ is a set of labeled vertices representing local transactions.[2]

3. γ is a set of arcs, each connecting two vertices in Γ.[3] A $\gamma - arc$ is formed when: Given two conflicting global transactions ($GT_i, GT_j \in \mathcal{GT}$), if an operation of any GST_i^k precedes a conflicting operation of GST_j^k in MH, a γ-arc is formed from GT_i to GT_j.

4. λ is a set of arcs, each connecting two vertices in $\Lambda \cup \Gamma$ when[4]:

   Given two conflicting transaction T_i^k and T_j^k submitted to $DBMS^k$ where T_i^k precedes T_j^k:

   (i) if $T_i^k, T_j^k \in \mathcal{LT}^k$ a $\lambda - arc$ is formed from T_i^k to T_j^k.

   (ii) if $T_i^k, T_j^k \in \mathcal{GST}^k$ a $\lambda - arc$ is formed between GT_i and GT_j respectively, i.e. $GT_i \rightarrow GT_j$.

   (iii) if $T_i^k \in \mathcal{GST}^k, T_j^k \in \mathcal{LT}^k$ (or vise versa) a $\lambda - arc$ is formed from GT_i to LT_j^k (or reverse: $LT_j^k \rightarrow GT_i$). □

The following definition states explicitly the meaning of λ-arcs in MSGs.

---

[1] Γ is formed from the earlier graph by redefining the vertices of V_1 such that for all vertices $GST_i^j \in GT_i$ in V_1, there is a single vertex $GT_i \in \Gamma$.

[2] Λ consists of precisely those vertices in V_2.

[3] If there is an arc in A_1 between two global subtransactions GST_i^k and GST_j^l, then an arc exists in γ between GT_i and GT_j.

[4] λ can be defined from A_2. The A_2 arcs are those which describe the interactions between global subtransaction in V_1 and local transaction in V_2. Since the set of vertices representing global subtransactions are collapsed into a single vertex in Γ this definition is a simplification of the earlier formulation.

Definition 4.2 *(λ-cycle)* A λ-cycle exists when a sequence of λ-arcs is formed that includes Γ and Λ nodes where all such Λ nodes exist at the same DBMS. □

Example 4.3 demonstrates how MSGs capture serialization of global and local transactions in a multidatabase history.

Example 4.3 Recall Examples 3.1 and 4.1. A history was described that was not serializable due to global transactions. This example demonstrates how MSGs model MHs and demonstrates that non-MDBSR schedules are captured by MSGs.

From Example 3.1 we can see that Γ and Λ have two vertices each, namely, $\{GT_1, GT_2\}$ and $\{LT_1^1, LT_1^2\}$, respectively. Since GT_1 conflicts with GT_2, we must determine if each global transaction's subtransactions precede the other in any of the local histories. This is determined by inspecting GH. $DBMS^1$ schedules $r_1^1(d)$ before $r_2^1(d)$ so an arc is required in γ from GT_1 to GT_2. $DBMS^2$ schedules $w_2^2(s)$ before $w_1^2(s)$ and since GT_1 and GT_2 conflict, a $\gamma - arc$ is required from GT_2 to GT_1. This creates the global cycle in MSG(MH). It is also necessary to consider the LHs. The local transactions cannot conflict with each other since they reference disjoint base sets so no arc exists between them. MH's global subtransactions do not conflict with each other or with local transactions so the set of $\lambda - arcs$ is empty (see Figure 4.2). Double headed arrows are used to represent $\gamma - arcs$ and single headed arrows for $\lambda - arcs$.◇

4.2 Multidatabase Serializability Theorem

MSGs are capable of illustrating precisely when an arbitrary multidatabase history is MDB-serializable. This section formally proves that cycle free MSGs are MDB-serializable. Before presenting the theorem it is useful to present a Lemma which describes how $\gamma - arcs$ can exist in a MSG for any history.

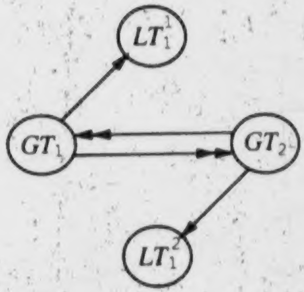


Figure 4.2: MSG for MH Described in Examples 3.1 and 4.1 - 4.3

Lemma 4.1 *(Mutual Exclusion of Global Subtransactions):* A $\gamma - arc$ can exist in a MSG between two global transactions GT_i and GT_j only if both global transactions submit a subtransaction to the same DBMS.

Proof: By definition (Definition 4.1), a $\gamma - arc$ can exist between GT_i and GT_j only if they conflict. Also, if GT_i and GT_j conflict, then they must have two subtransactions that conflict, which we will denote by GST_i^k and GST_j^l. These two subtransactions can conflict only if they have two operations, say $p \in GST_i^k$ and $q \in GST_j^l$, that conflict. By definition (Definition 2.1), p and q can conflict if they access the same data. However, according to the computational model described in Chapter 3, if p and q access the same data item, then they must be accessing the same database. Thus, $k = l$ for the subtransactions GST_i^k and GST_j^l to conflict.□

One final comment is necessary before presentation of the theorem. Multidatabase serializability is the combination of two types of serializable histories in a multidatabase history. When each local database scheduler produces a serializable history and the set of committed global transactions are globally serializable, the MDMS is said to have produced an MDB-serializable schedule. This is the same as ensuring that the MDB history is $\lambda - acyclic$ or that for each local history it is equivalent to some serial schedule, and that the MDB schedule is $\gamma - acyclic$ or the global history is equivalent to some serial ordering. Therefore, the proof process is simplified because each type of transaction can be considered separately.

Theorem 4.1 *(MDB Serializability Theorem)* A multidatabase history (MH) is MDB-serializable if and only if MSG(MH) is both $\gamma-acyclic$ and $\lambda-acyclic$.

Proof:

(if): Given a $\gamma-acyclic$ and $\lambda-acyclic$ MSG for a multidatabase history MH; MH is MDB-serializable. The proof is accomplished in two parts.

$\lambda-acyclic$: Since each DBMS produces only serializable schedules, λ-cycles at a specific DBMS are not possible. Further, the data is not replicated, so λ-arcs are not formed between transactions at different DBMSs. Therefore, λ-cycles are not possible.

$\gamma-acyclic$: Without loss of generality, assume that $MH =< \mathcal{LH}, GH >$ refers to the committed projection of a multidatabase history.[5] Consider the global history GH defined over the set of transactions $\mathcal{GT} = \{GT_1, \cdots, GT_n\}$. Without loss of generality, assume that the committed history (C(GH)) is $\{GT_1, \cdots, GT_m\}$. The $\Gamma-vertices$ of MSG(MH) ($\{GT_1, GT_2, ..., GT_n\}$) are $\gamma-acyclic$ so they can be topologically sorted with respect to $\gamma-arcs$. Let the permutation $i_1, i_2, ..., i_n$ be a permutation of $1, 2, ..., n$ such that $GT_{i_1}, GT_{i_2}, ..., GT_{i_n}$ is a topological sort of the $\Gamma-vertices$ of MSG(MH). Let GH_S be the serial history of $GT_{i_1}, GT_{i_2}, ...GT_{i_n}$. We will prove that: $GH \equiv GH_S$. Let $p \in GT_i$ and $q \in GT_j$ and p and q conflict such that $p \prec_{GH} q$. This means that there is a $\gamma-arc$ $GT_i \rightarrow GT_j$ in MSG(MH). Therefore, in any topological sort of GH, GT_i precedes GT_j. Thus, all operations of GT_i precede all operations of GT_j in any topological sort. Thus $GH \equiv GH_S$. Since GH_S is MDB-Serial, GH is MDB-Serializable.

(only if): Given that the history is MDB-serializable we will show that the MSG produced must be both $\gamma-acyclic$ and $\lambda-acyclic$.

---

[5] C(MH) is the committed history of a MDB schedule which includes the committed transaction in each local history and the global history. C(MH) includes $\{C(LH^1), C(LH^2), \cdots, C(LH^n)\}$ and C(GH) which are those GTs in GH that are committed.

First note that the set of $\lambda - arcs$ is subdivided into a number of disjoint subsets, each for one LH. Assume that a cycle exists in one of the subsets of $\lambda - arcs$ as follows: $T_i \rightarrow \cdots \rightarrow T_n \rightarrow \cdots \rightarrow T_i$. This implies that an operation of T_i precedes and conflicts with an operation of T_n and that an operation of T_n precedes and conflicts with an operation of T_i. This means that the DBMS which has generated the particular local history has incorrectly scheduled its transaction, which contradicts the assumption that all local schedulers function correctly. Thus, $\lambda - cyclicity$ cannot occur in a MDB-serializable history.

Suppose MH is serializable. Let MH_S be a serial history equivalent to the MDB-serializable history MH. Consider a $\gamma - arc$ $(GT_i \rightarrow GT_j) \in MSG(MH)$. This means that there exist two conflicting operations $p \in GT_i$ and $q \in GT_j$ such that $p \prec q$ in some local history (LH^k). This is true since, according to Lemma 4.1, both of these operations execute on the same database. Since MH_S is equivalent to MH and there is an arc from $GT_i \rightarrow GT_j$, all operations of GST_i^k occur before those of GST_j^k. Suppose there is a $\gamma - cycle$ in MSG(MH). This implies that there exists a $DBMS^l$ which scheduled an operation $r \in GT_j$ before an operation $s \in GT_i$. Since this implies that $GT_j \prec_{GH} GT_i$ in MH_S, an operation of GT_j precedes any of GT_i's. But, an operation of GT_i is known to precede an operation of GT_j at $DBMS^k$, which is contradictory. □

4.3 Comparison to Other Serializability Criteria

Two correctness criteria are generally accepted for serializability in transaction management systems: namely, view and conflict. *View serializability* requires that each transaction is guaranteed a consistent view of the database. Determining whether a history is view serializable is known to be computationally expensive. *Conflict serializability* is generally accepted as a correctness criterion for transactions in a database management system. All transactions are considered and conflicting operations are

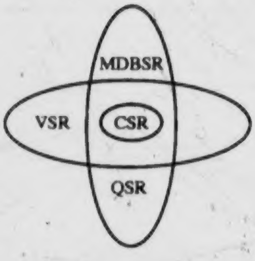


Figure 4.3: QSR and MDBSR Related to VSR and CSR

related by a partial order. A history is conflict serializable if and only if the history produces an acyclic serialization graph (SG) [3]. If we let VSR and CSR denote the set of histories that are view and conflict serializable, respectively, then CSR $\subset$ VSR [35]. This section establishes the relationship of the set of histories that are MDB-serializable (denoted as MDBSR) with CSR and VSR. We accomplish this by considering MDBSR's relationship to the set of histories that are quasi-serializable (QSR) [10]. It has been shown that QSR $\supset$ CSR and that QSR $\not\supset$ VSR nor is QSR $\not\subset$ VSR. This relationship is depicted in Figure 4.3.

Quasi-serializability and MDB-serializability are similar in construct and represent the same class of histories. A global history is quasi-serial if all local histories are (conflict) serializable and there exists a total order of all global transactions such that, for every two global transactions GT_i and GT_j where GT_i precedes GT_j in the order, all of GT_i's operations precede GT_j's operations in all local histories in which they both appear. A history is quasi-serializable if it is equivalent to a quasi-serial history. Quasi serialization graphs are used to model such histories. A "... global history H, denoted QSG(H), is a directed graph whose vertices are the global transactions in H, and whose arcs are all the relations (G_i, G_j) $(i \neq j)$ such that $G_i \rightarrow^H G_j$." [10]

Example 4.4 The previously described multidatabase system is used in this example. However, two different global transactions are submitted:

$GT_1 : w_1(d); r_1(t); c_1;$ $\qquad$ $GT_2 : r_2(e); r_2(s); w_2(u); c_2;$

The following local transactions are submitted:

$$LT_1^1 : \hat{r}_1^1(d); \hat{w}_1^1(e); \hat{c}_1^1; \qquad LT_1^2 : \hat{w}_1^2(t); \hat{r}_1^2(u); \hat{c}_1^2;$$

Assume that these orderings occur:

$$LH^1 : w_1(d); \hat{r}_1^1(d); \hat{w}_1^1(e); r_2(e); c_1; \hat{c}_1^1$$
$$LH^2 : r_2(s); \hat{w}_1^2(t); r_1(t); w_2(u); \hat{r}_1^2(u); c_2; \hat{c}_1^2;$$

To compare QSR with MDBSR we extend this example to encompass the activities of global subtransactions.

The global transactions GT_1 and GT_2 are divided into the following global subtransactions:

$$GST_1^1 : w_1^1(d); c_1^1; \qquad GST_1^2 : r_1^2(t); c_1^2;$$
$$GST_2^1 : r_2^1(e); c_2^1; \qquad GST_2^2 : r_2^2(s); w_2^2(u); c_2^2;$$

which produces the following equivalent histories:

$$LH^1 : w_1^1(d); \hat{r}_1^1(d); \hat{w}_1^1(e); r_2^1(e); c_1^1; \hat{c}_1^1; c_2^1;$$
$$LH^2 : r_2^2(s); \hat{w}_1^2(t); r_1^2(t); w_2^2(u); \hat{r}_1^2(u); c_2^2; \hat{c}_1^2; c_1^2;$$

The SG depicted in Figure 4.4 (a) is the serialization graph for the history. Although the history should be considered correct the serialization is cyclic so is not an acceptable serialization. This is too restrictive. However, QSR produces the QSG in Figure 4.4 (b) and MDBSR produces the MSG depicted in Figure 4.4 (c). ◇

The relationship between quasi-serializability and MDB-serializability is significant. Both correctness criteria attempt to capture the ordering of global transactions on a MDS. Quasi-serializability is concerned with global transaction orderings across

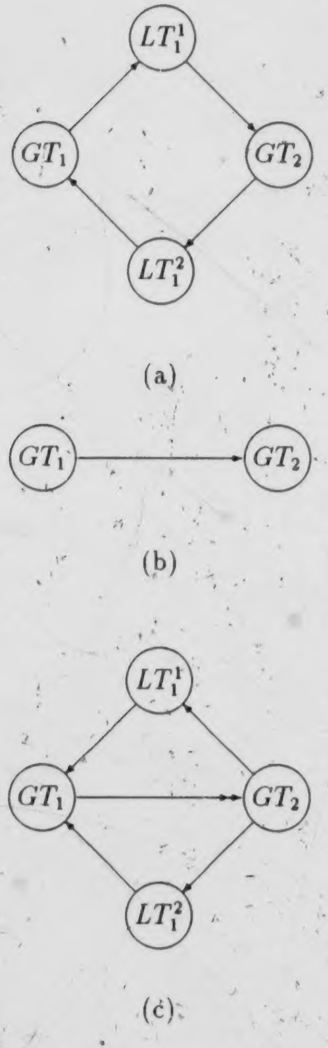

LT_1^1
GT_1
GT_2
LT_1^2
(a)
GT_1
GT_2
(b)
LT_1^1
GT_1
GT_2
LT_1^2
(c)

Figure 4.4: Sample Correctness Graphs: (a) SG (b) QSG (c) MSG

participating databases while assuming that local transactions are ordered correctly. MDB-serializability captures the activities of global transactions in the same way as QSR by identifying the activities of global subtransactions at each database explicitly.

The work on the MDB-serializability correctness criterion occurred separate from but in parallel with QSR. The similarities between the two are a result of a similar underlying philosophy. Both methods successfully capture the serialization of transactions in an MDB environment as demonstrated by the following theorem.

Theorem 4.2 QSR ≡ MDBSR.

Proof: The proof requires that we show that all histories of QSR are MDBSR and all histories are MDBSR that are also QSR. Since a history is QSR if and only if its QSG is acyclic and a history is MDBSR if and only if its MSG is acyclic so it is necessary to demonstrate that QSGs are equivalent to MSGs. The proof of the theorem follows directly from the definitions. Two equivalencies must be demonstrated to complete the proof: (1) local histories must be modeled equivalently and (2) global histories must be modeled equivalently.

Local histories which are QSR are assumed to be serializable, which means that all local histories must form an acyclic serialization graph. Local histories in MDBSR histories [illegible] also assumed to be serializable which means that all local histories must form an acyclic serialization graph. Since both form equivalent serialization graphs for local histories the first step of the proof is demonstrated. Intuitively, the SG for each local history is captured by $\lambda - arcs$ in MSGs.

Global histories are modeled by QSGs and MSGs in the same way. QSGs form an arc precisely when MSGs form $\gamma - arcs$. Since $\gamma - arcs$ and QSG arcs are formed identically so both graphs must capture the same histories equivalently, which satisfies the second requirement of the proof. □

4.4 MDB Schedulers

Scheduling of transactions in a MDS must be accomplished at both the global and local levels. Since we assume that each DBMS is capable of generating serializable local histories, the only requirement of the MDMS is to submit global subtransactions to each DBMS so that any local ordering can only produce a MDB-serializable schedule. Alternatively, the MDMS may submit transactions to each DBMS optimistically, provided the MDMS is capable of backing out of erroneous states which occur as a result. We consider schedulers of the former type, since the latter implies semantic knowledge about transactions or restricts the use of the schedulers to those application domains where compensating transactions can be used.

Scheduling global transactions in a serializable order with respect to each other is not sufficient. The following example demonstrates the additional difficulties introduced by autonomous local scheduling of transactions.

Example 4.5 Consider the following two local databases: $\mathcal{LDB}^1 = \{d, e, f, g\}$ and $\mathcal{LDB}^2 = \{s, t, u, v\}$. Two global transactions are submitted:

$$GT_1 : r_1(d); r_1(s); c_1;$$
$$GT_2 : r_2(f); r_2(u); c_2;$$

producing the following GSTs:

$$GST_1^1 : r_1^1(d); c_1^1; \quad GST_1^2 : r_1^2(s); c_1^2;$$
$$GST_2^1 : r_2^1(f); c_2^1; \quad GST_2^2 : r_2^2(u); c_2^2;$$

The following two LTs are submitted to the DBMSs:

$$LT_1^1 : \hat{w}_1^1(d); \hat{w}_1^1(f); \hat{c}_1^1;$$
$$LT_1^2 : \hat{w}_1^2(s); \hat{w}_1^2(u); \hat{c}_1^2;$$

which generate the histories:

$$LH^1 : r_1^1(d); \dot{w}_1^1(d); \dot{w}_1^1(f); r_2^1(f); c_1^1; \dot{c}_1^1; c_2^1;$$
$$LH^2 : r_2^2(u); \dot{w}_1^2(s); \dot{w}_1^2(u); r_1^2(s); c_2^2; \dot{c}_1^2; c_1^2;$$

LH^1's scheduler produced a locally correct serial execution such that $GT_1 \prec LT_1^1 \prec GT_2$. Similarly, LH^2's scheduler produced the correct serial order $GT_2 \prec LT_1^2 \prec GT_1$. Taken together, however, it is evident that these transactions have been incorrectly serialized. ◇

From this example a number of things become apparent:

1. All operations in global subtransactions must be assumed to conflict if they are submitted to the same DBMS at the same time.
2. Global transactions which access mutually disjoint base sets may still conflict due to local transactions. This occurs because each subtransaction is dependent upon the orderings of other related subtransactions. Recall that these are known as indirect conflict [10].
3. Detecting conflicts based on operations and base sets of transactions is not possible, so solutions which do not use semantic information but attempt to provide "better than serial concurrency" (at an operation level), are unlikely to prove successful.

The difficulty disclosed by Example 4.5 is that two local schedulers may make different decisions. Unfortunately, there is no way to detect such occurrences at the MDMS layer until after the execution, even though both global transactions are read-only and access mutually disjoint base sets. This implies that semantic information must be present to increase concurrency beyond serial execution.

The algorithms presented avoid this difficulty by accepting serial global subtransaction executions as the best possible submission policy while executing global transactions concurrently. These algorithms achieve concurrency without placing restrictions on the formation of the global subtransactions.

Before describing the schedulers we list possible states of a transaction. Transactions can be in one of three possible states:

Active: LTs and GSTs are active if they have been submitted to a DBMS. GTs are active if at least one global subtransaction has been submitted on its behalf to a DBMS but all GSTs have not terminated.

Passive: A transaction is passive if it has been suspended in a queue. All types of transactions may be passivated. A global transaction can only be passive if it is suspended as a unit by the global scheduler. LTs and GSTs are passive if the entire transaction is blocked by a local scheduler.

Complete: LTs and GSTs complete at the time they commit or abort. A GT is complete only after all of its GSTs have been completed. The global scheduler is then responsible for determining a completion status (abort or commit) for each GT.

4.4.1 Global Serial Scheduler

Example 4.5 demonstrated the difficulty involved in operation level scheduling of global transactions by a global scheduler. Therefore, the global scheduler uses a global subtransaction as the smallest unit schedulable. To motivate the algorithm an example is presented that demonstrates intuitively when global subtransactions can be safely submitted to their respective DBMSs.

Example 4.6 Since access to any data item in an LDB can cause a non-MDB-serializable schedule, the actual operations of the GTs and their GSTs is independent of the MDMS's scheduling algorithm. This example uses five global transactions, submitted to the MDMS, each arriving in the indicated sequence and requiring the following GSTs.

$$\begin{array}{llll}
GT_1 : & GST_1^1 & & \\
GT_2 : & GST_2^1 & GST_2^2 & \\
GT_3 : & GST_3^1 & GST_3^2 & GST_3^3 \\
GT_4 : & & GST_4^2 & GST_4^3 \\
GT_5 : & & & GST_5^3
\end{array}$$

Assume that no other GTs are active or passive when GT_1 is submitted to the MDMS. The following scheduling can occur as each GT is submitted.

GT_1 arrives and since only one global subtransaction is required to perform all of the operations of GT_1, no scheduling difficulties will occur if GST_1^1 is submitted immediately to $DBMS^1$.

GT_2 is submitted and parsed into two GSTs which are to be submitted to $DBMS^1$ and $DBMS^2$. Assume, for the moment, that both of these GSTs are submitted to their respective DBMSs. Consider the possible schedules: GST_2^2 can be readily submitted to $DBMS^2$ since there are no other GSTs currently active at the DBMS. Two orderings are possible at $DBMS^1$ because two GTs would have GSTs at that DBMS. If GST_1^1 is scheduled before GST_2^1 by $DBMS^1$, it is the same as scheduling GT_1 before GT_2 in a global serial order. If GST_2^1 is scheduled before GST_1^1, the schedule produced would be equivalent to scheduling GT_2 before GT_1. Both of these schedules maintain consistency so GT_2 can be submitted to the DBMSs.

GT_3 is submitted and parsed into three GSTs. Since GT_1 has only one GST it will not cause the suspension of GT_3, for the same reason cited in the scheduling of GT_2. However, GT_2 accesses more than a single database and more importantly, GT_2's global subtransactions access two of the same databases that GT_3 needs to access (namely $DBMS^1$ and $DBMS^2$). Assume that GT_3 is permitted to proceed and all of the GSTs are submitted to their required DBMSs. GST_3^3 should not cause any difficulty; however, GST_3^1 and GST_3^2 can cause incorrect schedules as follows: If GST_2^1 is executed before GST_3^1 at $DBMS^1$ but GST_3^2 is executed before GST_2^2 at $DBMS^2$, the implication is that $GT_1 \prec GT_2$ and that $GT_2 \prec GT_1$, which is erroneous. Therefore, GT_3 must be passivated.

GT_4 is submitted and parsed into two GSTs. Since GT_3 is passive, GT_4 must be analyzed with respect to GT_2. GST_4^3 could be submitted without ordering difficulties. Consider GST_4^2. Assume that GST_2^2 is executed before GST_4^2 which implies that $GT_2 \prec GT_4$ at all DBMSs. Alternatively, if GST_4^2 is executed before GST_2^2, then $GT_4 \prec GT_2$ at all DBMSs. Since either ordering is acceptable and these are the only possible orderings, the GSTs of GT_4 can be submitted to the DBMSs.

GT_5 is submitted and parsed into a single GST. Even though other GTs are present any sequencing of the GTs is MDB-serializable so the GST can be submitted.

Assume that GTs, and their GSTs terminate in the same order they were submitted. When GT_1 terminates, GT_3 must remain blocked because GT_2 is still active. When GT_2 completes (after both GST_2^1 and GST_2^2 complete) GT_3 must be retested to determine if the set of GSTs required by GT_3 can be submitted. Since GT_4 is active and accesses more than a single DBMS to which GT_3 desires access, it is necessary to leave GT_3 passive. Once GT_4 completes; GT_3's global subtransactions can be submitted without difficulty. ◇

Two additional examples will demonstrate the other conditions necessary for the correct submission of global subtransaction.

Example 4.7 Consider the following sequence of global transactions and their corresponding GSTs :

$$\begin{array}{llll} GT_1 : & GST_1^1 & GST_1^2 & \\ GT_2 : & & GST_2^2 & GST_2^3 \\ GT_3 : & GST_3^1 & & GST_3^3 \end{array}$$

GT_1 and GT_2 can be submitted in an obvious way. Consider the submission of GT_3. If it was permitted to execute the following sequence is possible:

$$\begin{array}{ll} DBMS^1 : & GT_3 \prec GT_1 \\ DBMS^2 : & GT_1 \prec GT_2 \\ DBMS^3 : & GT_2 \prec GT_3 \end{array}$$

which would produce a γ-cyclic MSG where $GT_1 \rightarrow GT_2 \rightarrow GT_3 \rightarrow GT_1$. □

In Example 4.7 it is necessary to block GT_3 until the completion of both GT_1 and GT_2.

Example 4.8 Consider Example 4.7. GT_3 must be blocked until the completion of GT_1 and GT_2. To see this consider GT_3's immediate submission. Further, assume GT_3 is submitted after GT_1 completes but while GT_2 is still active. This implies that $GT_1 \rightarrow GT_2$ exists in the MSG. The ordering of GT_2 and GT_3 at $DBMS^3$ is undefined so either $GT_2 \rightarrow GT_3$ or $GT_3 \rightarrow GT_2$ will exist in the MSG. At this time, the ordering of GST_1^2 and GST_2^2 is unknown so a γ-cycle is possible if GT_3 is submitted. □

Examples 4.6 to 4.8 suggest the conditions under which GTs can be submitted so any ordering of transactions at the DBMSs will produce only MDB-serializable schedules. Each condition is stated below:

1. Global transactions which access a single database and require only a single global subtransaction can be submitted immediately. Such submissions are possible because conflicts between databases are impossible and the local schedulers guarantee correct local scheduling. Note that Definition 3.3 describes such transactions as those which accesses remote databases.

2. Global transactions which access more than one database, but no other global transactions are currently accessing those databases, can have their GSTs submitted without delay.

3. If a global transaction accesses multiple databases, all of which are concurrently accessed by GTs that only access a single database, then the global subtransactions can be submitted.

4. If a global transaction accesses multiple databases and other active global transactions only access one of these databases, then the global transaction's GSTs can be submitted.

5. A global transactions is blocked if more than one of the databases accessed is being accessed by others which are active.

Finally, the blocked transaction queues must be serviced in a first-in, first-out fashion. This will avoid transaction starvation.

The *Global Serial Scheduler* (GSS) algorithm produces MDB-serializable schedules. Scheduling of GSTs with the GSS uses an "all or nothing" approach whereby all of a global transaction's subtransactions are submitted or it is passivated.

Global Serial Scheduler

(Initial GT Submission)

1. A global transaction is parsed into a set of global subtransactions.

2. If there is only one global subtransaction required by the global transaction, it is submitted to its DBMS and the scheduler terminates.

3. If multiple subtransactions are required, the global scheduler determines if there are active global transactions which access more than one of the databases accessed by the GT being scheduled. If there is such a set of active global

transactions, the GT being scheduled is passivated until a global transaction completes.

4. If no overlapping global transactions are present, the global subtransactions are submitted.

The scheduler only describes the submission of new global transactions to the MDMS. As GSTs complete, it is necessary to update the current status of active transactions at the DBMSs and to activate passive global transactions when possible. The following describes the actions necessary when a global subtransaction completes.

Global Serial Scheduler

(*GST completion process*)

1. When a global subtransaction completes that is not the last subtransaction required by a global transaction, the completion is recorded and the algorithm terminates.
2. If this is the last subtransaction to complete, waiting global transactions are retested to determine if they can be submitted.
3. For each waiting global transaction, determine if any active global transactions access more than one of the required DBMSs. If such global transactions are present, the tested GT is requeued and the next one is tested. If no such global transactions exist, all of global transaction's subtransactions are submitted and it is removed from the queue before the next one is tested.

The following is a list of function/data structures necessary for the description of the algorithm:

makesubs(*GT*) : A function which takes a global transaction submitted to the MDMS and returns a set of GSTs. This set of GSTs are saved in the internal data structures necessary to manage the GT.

DBMS_set(GT_i) : The set of DBMSs where the subtransactions for global transaction GT_i are executed.

Active_set($DBMS^k$) : The set of global transactions which have an active global subtransaction executing at $DBMS^k$.

Wait_Q : Global transactions which cannot be submitted immediately are placed on this queue.

conflict_set (GT_i) : Set of DBMSs where GT_i may conflict.

card : The cardinality function returns the number of elements in the argument set.

Figures 4.5 and 4.6 present the algorithm. Global transactions submitted to the MDMS are processed in a slightly different way initially. Figure 4.5 describes the process required when a GT is submitted. Each step of the intuitive argument is reflected in this detailed presentation. Line (1) creates the GSTs and line (2) forms a set identifying the DBMSs accessed by the GT. Line (3) handles the case where only one DBMS is accessed by the GT. This requires that the Active_set for that DBMS be updated to reflect the submission of the GST (line (5)). Line (6) actually submits the GST. Global transactions which access more than a single database must be tested to determine if there is an overlap with active GT's global subtransactions. The set of currently active GTs is produced (line (7)) and any global transactions which only access a single DBMS are removed (line (8)). The set of active transactions (Active_GTs) is used to determine the set of DBMSs which could cause the GT under consideration to be passivated (line (9)). If GTs exist whose combined conflict_sets overlap, the global transaction is passivated (line (16)). If the GSTs can be submitted the conflict_sets for each active GT must be updated (line (10–11)) and the conflict_set for the GT being submitted must be created (line (12)). Finally, the global subtransactions must be activated by updating the Active_set for each DBMS and submitted (line (14–15)).

Algorithm 4.1 *(Global Serial Scheduler - Initial Scheduling)*

```
begin
  input $GT_i$ : global transaction;

  var   Active_GTs : set of global transactions

  $\mathcal{GST}_i \leftarrow$ makesubs($GT_i$);
  DBMS_set($GT_i$) ← set of DBMSs accessed;

  if card(DBMS_set ($GT_i$)) = 1 then
    begin
      k ← database accessed;
      Active_set($DBMS^k$) ← Active_set($DBMS^k$) ∪ $GT_i$;
      submit $GST_i^k$ to $DBMS^k$
    end
  else begin
      Active_GTs ← $\bigcup_k$ Active_set ($DBMS^k$);
      Active_GTs ← $\{GT_k \mid GT_k \in$ Active_GTs $\wedge$ card $(\mathcal{GST}_k) > 1\}$
      if card(DBMS_set($GT_i$) $\cap(\bigcup_{GT_j \in Active\_GTs}$ conflict_set($GT_j$))) $\leq 1$)
        then begin
          for each $GT_j \in$ Active_GTs do
            conflict_set($GT_j$) ← conflict_set ($GT_j$)∪ DBMS_set ($GT_i$);
          conflict_set ($GT_i$) ← $\bigcup_{GT_j \in Active\_GTs}$ conflict_set($GT_j$)∪
              DBMS_set ($GT_i$)
          for each $DBMS^k \in$ DBMS_set ($GT_i$) do
            begin
              Active_set ($DBMS^k$) ← Active_set ($DBMS^k$) ∪ $GT_i$;
              submit $GST_i^k$ to $DBMS^k$;
            end
          endfor
        end
      else
        passivate $GT_i$ on Wait_Q;
      endif
    end
  endif
end
```

Figure 4.5: Global Serial Scheduler (Part 1)

Completion of the GST requires some housekeeping to ensure that the completion is noted by the global scheduler (See Figure 4.6). Since the GT is no longer active at the DBMS, it must be removed from the Active_set at Line (1) and from the global transaction set (Line (2)). If the entire global transaction has completed (Line (3)), waiting subtransactions can be scheduled, if possible. The difference between this and the initial arrival of the GT is that the entire queue of waiting global transaction must be tested so an extra loop is required (Line (4)). Testing of each routine is identical to the initial arrival except when the GT must be passivated again. In the event that the GT must be passivated it is tagged, so that an infinite loop does not occur from replacing it in the queue. Hence the need to **repassivate** at Line (14).

The next example provides a detailed trace of the GSS's execution.

Example 4.9 Five global transactions are to be submitted to a MDMS using the Global Serial Scheduler (GSS) Algorithm. The GSTs required are the same as those used in Example 4.6.

Each GT will be discussed in turn:

1. GT_1 is submitted to the GSS and accesses only a single DBMS. The first **if** test (line (3)) passes since the cardinality of DBMS_set(GT_1) = $\{DBMS^1\} = 1$. The DBMS accessed is $DBMS^1$ so the Active_set($DBMS^1$) = $\{GT_1\}$ and the subtransaction is submitted to $DBMS^1$. See Figure 4.7 line (1) which describes the Active_sets and summarizes the DBMS_sets for this and subsequent global transaction.

2. GT_2 is submitted and requires two subtransactions accessing $DBMS^1$ and $DBMS^2$. The initial **if** (line (3)) fails so the **else** (line (7)) is executed. Since GT_1 is the only other active GT and it only accesses a single DBMS, the intersection of the DBMS_sets cannot be greater than 1, so the condition passes and GT_2 is submitted. All of GT_2's GSTs are submitted but the Active_sets for $DBMS^1$ and $DBMS^2$ must be updated (Line (2) of Figure 4.7).

Algorithm 4.1 *(Global Serial Scheduler - Subtransaction Termination)*

```
begin
  input   $GST_i^k$ : GST for $GT_i$ at $DBMS^k$ completes;

  var     Active_GTs : set of global transactions

  Active_set ($DBMS^k$) $\leftarrow$ Active_set ($DBMS^k$) $- GT_i$;                    (1)
  DBMS_set ($GT_i$) $\leftarrow$ DBMS_set ($GT_i$) $- DBMS^k$;                          (2)
  if DBMS_set ($GT_i$) $= \phi$ then                                                   (3)
    for each $GT_n$ in Wait_Q do                                                       (4)
      begin
        Active_GTs $\leftarrow \bigcup_k$ Active_set ($DBMS^k$);                        (5)
        Active_GTs $\leftarrow \{GT_k \mid GT_k \in$ Active_GTs $\wedge$ card $(\mathcal{G}ST_k) > 1\}$   (6)
        if card(DBMS_set($GT_n$) $\cap$ ($\bigcup_{GT_j \in Active\_GTs}$ conflict_set($GT_j$))) $\leq 1$)   (7)
          then begin
            for each $GT_j \in$ Active_GTs do                                          (8)
              conflict_set($GT_j$) $\leftarrow$ conflict_set ($GT_j$)$\cup$ DBMS_set ($GT_n$);   (9)
            conflict_set ($GT_i$) $\leftarrow \bigcup_{GT_j \in Active\_GTs}$ conflict_set($GT_j$)$\cup$
                DBMS_set ($GT_n$)                                                      (10)

            for each $DBMS^k \in$ DBMS_set ($GT_n$) do                                 (11)
              begin
                Active_set ($DBMS^k$) $\leftarrow$ Active_set ($DBMS^k$) $\cup GT_n$;   (12)
                submit $GST_n^k$ to $DBMS^k$;                                          (13)
              end
            endfor
          end
        else
          repassivate $GT_i$ on Wait_Q;                                                (14)
        endif
      end
    endfor
  endif
end
```

Figure 4.6: Global Serial Scheduler (Part 2)

3. GT_3 is divided into three subtransactions that must be submitted to all three DBMSs. The **if** (line (9)) only needs to check:

$$\begin{aligned}\text{DBMS\_set}(GT_2) \;\cap\;& \text{DBMS\_set}(GT_3)\\ =\;& \{DBMS^1, DBMS^2\} \cap \{DBMS^1, DBMS^2, DBMS^3\}\\ =\;& \{DBMS^1, DBMS^2\}\end{aligned}$$

   Since the cardinality of this set is 2, the test fails so none of GT_3's GSTs can be submitted and GT_3 must be placed on the Wait_Q. Note that the Active_sets remain the same and the Wait_Q now contains GT_3 (line (3) Figure 4.7).

4. GT_4 is submitted to the GSS and requires service from $DBMS^2$ and $DBMS^3$. The **if** line (9) succeeds because DBMS_set(GT_1) $\cap \bigcup_{GT_2}$ conflict_set(GT_2) $= \phi$ and DBMS_set(GT_2) $\cap$ DBMS_set(GT_4) $= \{DBMS^2\} \not> 1$ so all GSTs can be submitted leaving the Active_sets as depicted in line (4) of Figure 4.7.

5. GT_5 is submitted, but since its DBMS_set contains only one GST the transaction can be scheduled without delay (line (5) Figure 4.7).

GT_3 remains blocked on the Wait_Q until active transactions complete. Assume that the GTs complete in the order they were submitted.

1. GST_1^1 completes and is removed from $DBMS^1$'s Active_set as indicated in line (6) of Figure 4.7. The identifier for $DBMS^1$ is removed from the DBMS_set of GT_1. Since DBMS_set (GT_1) $= \phi$ the Wait_Q is processed. Each GT waiting in the queue is retested against the current Active_sets. Since GT_3 is the only waiting transaction and GT_2 is still active, the test fails and GT_3 is repassivated on the Wait_Q.

2. GST_2^1 completes so GT_2 is removed from Active_set ($DBMS^1$) and $DBMS^1$ is removed from the DBMS_set(GT_2) leaving $\{DBMS^2\}$. Since Active_set(GT_2)

$\neq \phi$ the process terminates without processing the Wait_Q. Line (7) of Figure 4.7 illustrates the current situation. GST_2^2 completes so GT_2 is removed from Active_set($DBMS^2$) and DBMS_set(GT_2) $= \phi$ so the Wait_Q is processed. Line (8) of Figure 4.7 illustrates the current status of the Active_sets. GT_3 is tested against the condition to determine if it can be submitted. Unfortunately,

$$\begin{aligned} DBMS\_set(GT_3) \cap \bigcup_{GT_4} \text{conflict\_set}(GT_4) &= \\ \{DBMS^2, DBMS^3\} \cap \{DBMS^1, DBMS^2, DBMS^3\} &= \\ \{DBMS^1, DBMS^2, DBMS^3\} & \end{aligned}$$

set's cardinality is greater than 1 so it is repassivated.

3. GST_4^2 completes, producing the situation in line (9) of Figure 4.7 but the Wait_Q is not processed since DBMS_set(GT_4) $= \{DBMS^3\}$. When GST_4^3 terminates, the DBMS_set(GT_4) $= \phi$ and the Wait_Q is processed. The only active transaction remaining is GT_5, executing at $DBMS^3$, so GT_3 can be scheduled leaving the Active_sets described in line (10) of Figure 4.7.

4. The balance of the subtransactions will terminate in an obvious way. ◇

Note that multiple global subtransactions have been submitted to the same DBMS but the schedules produced are MDB-serializable as proven in the next section.

4.4.2 Correctness of the Global Serial Scheduler

Proving the correctness of the GSS is accomplished in two steps. First, the characteristics of histories produced by the GSS are identified. Second, it is proven that any history which has these properties is MDB-serializable.

	Active_set($DBMS^1$)	Active_set($DBMS^2$)	Active_set($DBMS^3$)
(0)	ϕ	ϕ	ϕ
(1)	$\{GT_1\}$	ϕ	ϕ
(2)	$\{GT_1, GT_2\}$	$\{GT_2\}$	ϕ
(3)	$\{GT_1, GT_2\}$	$\{GT_2\}$	ϕ
(4)	$\{GT_1, GT_2\}$	$\{GT_2, GT_4\}$	$\{GT_4\}$
(5)	$\{GT_1, GT_2\}$	$\{GT_2, GT_4\}$	$\{GT_4, GT_5\}$
(6)	$\{GT_2\}$	$\{GT_2, GT_4\}$	$\{GT_4, GT_5\}$
(7)	ϕ	$\{GT_2, GT_4\}$	$\{GT_4, GT_5\}$
(8)	ϕ	$\{GT_4\}$	$\{GT_4, GT_5\}$
(9)	ϕ	ϕ	$\{GT_4, GT_5\}$
(10)	$\{GT_3\}$	$\{GT_3\}$	$\{GT_3, GT_5\}$

DBMS_set

$$(GT_1) = \{DBMS^1\}$$
$$(GT_2) = \{DBMS^1, DBMS^2\}$$
$$(GT_3) = \{DBMS^1, DBMS^2, DBMS^3\}$$
$$(GT_4) = \{DBMS^2, DBMS^3\}$$
$$(GT_5) = \{DBMS^3\}$$

Figure 4.7: Trace of Scheduler Execution for Example 4.3

In the computational model we assume that each DBMS is capable of correctly serializing all transactions submitted. This is stated in the following Proposition.

Proposition 4.1 Each local scheduler always schedules all transactions in a serializable order. □

The global scheduler does not control how each operation is scheduled but does determine when global subtransactions are submitted. The GSS schedules a global transaction's subtransactions in an all or nothing fashion.

Proposition 4.2 The GSS algorithm either submits all of a global transaction's subtransactions or none of them. □

This seen in lines (8-10) of Figure 4.5 and lines (6-8) of Figure 4.6. A global transaction's subtransactions are submitted only if no other global transaction is active which accesses more than one common database. If such a global transaction exists, the subtransactions must wait until all such global transactions have completed. This guarantees that all global subtransactions are executed MDB-serializably. The following Lemma proves this.

Lemma 4.2 For every pair of global transactions $GT_i, GT_j \in \mathcal{GT}$, scheduled by the GSS, either all of GT_i's subtransactions are executed before GT_js at every DBMS or vice versa.

Proof: (Case 1): A global transaction GT_i requiring only a single subtransaction (GST_i^k) is always scheduled immediately (line (3) Figure 4.5). From Proposition 4.1 it follows that any other $GST_j^k \in GT_j$ either precedes or follows $GST_i^k \in GT_i$.

(Case 2): A global transaction GT_i which submits to multiple DBMSs but only accesses one DBMS accessed by all other active global transactions is submitted immediately (see line (9) Figure 4.5 and line (7) Figure 4.6). Let $\mathcal{CGT}$ be the set of active global transactions. Without loss of generality, consider $GST_i^k \in GT_i$ and $GST_j^k \in GT_j$ ($GT_j \in \mathcal{CGT}$) where the global transactions overlap only at $DBMS^k$. Since $DBMS^k$ is the only place where GT_i and GT_j occur together, we know from Proposition 4.1 that either $GT_i \prec GT_j$ or $GT_j \prec GT_i$, as required.

(Case 3): A global transactions which submit to multiple DBMSs but accesses more than one DBMS accessed by active global transactions are blocked (see line (9) & (16) of Figure 4.5 and line (7) & (14) of Figure 4.6). Such a global transaction is blocked in its entirety (see Proposition 4.2) until the completion of the offending global transactions (see line (3) of Figure 4.6) when it is scheduled if cases (1) or (2) hold. □

We have characterized the type of histories produced by the GSS. The second step is to demonstrate that all such histories are MDB-serializable. Recall that all MDB-serializable histories produce acyclic MSGs.

Theorem 4.3 The Global Serial Scheduler produces only MDB-serializable histories.

Proof: Since all MDB-serializable histories can be represented by $\lambda - acyclic$ and $\gamma - acyclic$ MSGs as demonstrated by Theorem 4.1, the proof is in two parts: (1) $\lambda - acyclicity$ is proven and (2) $\gamma - acyclicity$ is proven.

$\lambda - acyclic$: $\lambda - cycles$ can occur in three possible ways. A $\lambda - cycle$ can occur between a local transaction and a local transaction; between a global transaction and a local transaction; or between a global transaction and a global transaction. Each case is dealt with in turn.

Between local transactions: Note that the Global Serial Scheduler does not schedule local transactions; in fact it does not have any effect on the local histories generated by the local transaction manager. Therefore, any $\lambda - cycles$ among local transactions must be caused by the local schedulers. However, this would indicate that the local scheduler creates nonserializable local histories, which is violates Proposition 4.1.

Between a local transaction and a global transaction: Again, if there is $\lambda - cycle$ involving a local transaction and a global subtransaction, this has to exist in a local history. As in the previous case, this would violate Proposition 4.1.

Between global transactions: Assume that a sequence of $\lambda - arcs$ exists from $GT_i \rightarrow T_{i+1} \rightarrow \cdots \rightarrow T_{n-1} \rightarrow GT_n$. This implies that an operation of a global subtransaction $GST_i^k \in GT_i$ conflicts and precedes some operation of $GST_n^k \in GT_n$ at some $DBMS^k$. We prove that the sequence of arcs $GT_n \rightarrow T_{n+1} \rightarrow \cdots \rightarrow T_m \rightarrow GT_i$ cannot exist. Two cases must be proven. (1) There exists a $GST_n^k \in GT_n$ which has an operation which precedes and conflicts with an operation of $GST_i^k \in GT_i$ at some $DBMS^k$. This contradicts Proposition 4.1. (2) There exists an operation of $GST_n^l \in GT_n$ which precedes and conflicts with an operation $GST_i^l \in GT_i$ at a different $DBMS^l$. This implies that the local scheduler at $DBMS^k$ scheduled GST_i^k before GST_n^k while $DBMS^l$ scheduled GST_n^l before GST_i^l. This contradicts Lemma 4.2.

$\gamma - acyclic$: Given that there is a sequence of $\gamma - arcs$ from $GT_i \rightarrow GT_{i+1} \rightarrow \cdots \rightarrow GT_{n-1} \rightarrow GT_n$. This implies that there exists an operation of $GST_i^k \in GT_i$ which precedes and conflicts with an operation of $GST_n^k \in GT_n$. Assume that a sequence of $\gamma - arcs$ exists such that $GT_n \rightarrow GT_{n+1} \rightarrow \cdots \rightarrow GT_m \rightarrow GT_i$. This sequence implies one of two possibilities. (1) An operation of $GST_n^k \in GT_n$ precedes and conflicts with an operation of $GST_i^k \in GT_i$ at $DBMS^k$. This contradicts Proposition 4.1. (2) An operation of some other $GST_n^l \in GT_n$ precedes and conflicts with an operation of $GST_i^l \in GT_i$ at another $DBMS^l$. This contradicts Lemma 4.2. □

4.4.3 Aggressive Global Serial Scheduler

Example 4.9 demonstrates the all or nothing principle followed by the GSS algorithm. It blocks the execution of a global transaction (GT_i) if there already exists other global transactions accessing more than a single DBMS accessed by GT_i. In Example 4.9, GT_3 is blocked because it must access all three local databases and potential MDB-serialization difficulties may occur at $DBMS^1$ and $DBMS^2$. Notice also that GT_3 must access $DBMS^3$ and there is currently no global transactions which has submitted (or is waiting to submit) a global subtransaction to $DBMS^3$. This suggests that GST_3^3 could be permitted to execute while GST_3^1 and GST_3^2 are blocked. In effect, such an algorithm would be more aggressive than the GSS, hence the *Aggressive Global Serial Scheduler* (AGSS) algorithm.

The algorithm is presented intuitively before the details are described. As with the GSS, the AGSS is presented in two parts; the first is the arrival of a new global transaction and second the completion of a global subtransaction.

Aggressive Global Serial Scheduler

(Initial GT submission)

1. Form the required global subtransactions for the global transaction.
2. If the global transaction has only a single global subtransaction, it can be submitted.
3. Each global subtransaction is scheduled independently. Each is submitted or suspended based on the activities of other active global transactions. All global subtransactions form a unique set of candidate GTs which may cause the GST to be suspended. The candidate set is based on the DBMS to which the GST is to be submitted. For example, given $GST_i^k \in GT_i$ to be submitted to $DBMS^k$, the candidate set is composed of all global transactions which are waiting to access data at LDB^k or which have an active GST at $DBMS^k$. In addition to these GTs, any GT which overlaps with them may cause non-MDB-serializable schedules. This set of global transactions could form non-MDB-serializable schedules so the entire set is considered when attempting to submit a global subtransaction.

   Given an arbitrary GT_j in the candidate set; the set of DBMSs currently accessed by GT_j is formed. If the intersection of this set and the set of DBMSs to be accessed is not empty, then the GST cannot be submitted so Step (4) is executed. Otherwise, Step (5) is performed.

4. The GST cannot be submitted immediately, so it is suspended in a wait queue and the next subtransaction is tested at Step (3). If no other GSTs need to be tested the algorithm terminates.

5. The global subtransaction is submitted. If there is another global subtransaction to be submitted, it is tested at Step (3). Otherwise, the algorithm terminates.

Completion of global subtransactions are handled differently in the AGSS than the GSS, since some GSTs may still be waiting for submission. The following describes GST completion.

Aggressive Global Serial Scheduler

(GSTs Completion Process)

1. The completion of the global subtransaction is recorded.

2. Completion of a global transaction may permit the submission of waiting subtransactions. This involves testing every waiting global subtransaction at each DBMS where the completing GT was active.

3. The global subtransaction at the head of the wait queue undergoes the same testing procedure as in Step (3) of the initial global transaction arrival process described above. If a GT is active which can cause a non-MDBSR schedule, Step (4) is performed, otherwise Step (5).

4. Since a GT is still present that could cause a non-MDBSR schedule the GST remains passive. The next GST waiting is tested at Step (3) unless the entire queue has been tested which terminates the process.

5. The GST can be submitted so its status is changed from passive to active. If GSTs are waiting, each must be retested at Step (3) or the algorithm terminates.

Some of the data structures used in the GSS algorithm are required by this algorithm. Since global subtransactions are submitted on a case by case basis it is necessary to modify the Wait_Q structure, but the balance of the required data structures remain the same. The following is a list of additional data structures required:

Wait_Q(*DBMS*) : Global subtransactions which cannot be submitted immediately are placed on a queue waiting to access the DBMS. There is one Wait_Q for each DBMS.

GSTs_active(GT_i) : Set of DBMSs which have an active global subtransaction submitted by GT_i.

GSTs_passive(GT_i) : Set of DBMSs which have passive or waiting subtransaction submitted by GT_i.

GSTs_complete(GT_i) : Set of DBMSs which have completed the global subtransactions submitted by GT_i.

Figure 4.8 and 4.9 present the algorithm in two parts. Global transactions are submitted to the MDMS and are processed in a different way initially. Figure 4.8 describes the GT initial submission process. Line (1) creates the GSTs and line (2) forms a set describing the DBMSs accessed by the GT. Line (3), (4) and (5) initialize the GTs reference sets to record the status of the GSTs. Line (6) handles the case where a GT only accesses one DBMS. The Active_set for that DBMS is updated to reflect the submission of the GST. Line (8) and (9) update the MDMS's status to indicate that the GST and the GT have been submitted, respectively. Line (10) actually submits the GST.

Global transactions accessing more than a single database are tested to determine if an overlap exists with other GTs. Each global subtransaction (line (11)) is independently tested for submissability. The set of global transactions which could cause the passivation of a global subtransaction are those that have a GST at or waiting to be submitted to that DBMS (line (12)). In addition to these GTs, any others which overlap with them must be included in the candidate set (line (13–15)). Any active GT which only accesses a single DBMS can be removed from this set (line (10)). Assume that $GST_i^k \in GT_i$ is to be submitted to $DBMS^k$. Line (17) determines if there exists another GT_j which has a GST_j^k waiting or active at $DBMS^k$, which also accesses another DBMS accessed by GT_i. If such a GT_j exists, GST_i^k is passivated (line (18–19)). Otherwise, GST_i^k may be submitted at line (20–22).

When a global subtransaction completes, the sets are updated to reflect the completion on Line (1–3) of Figure 4.9. If this is the final GST to complete the transaction completion sequence is initiated at Line (5). Finally, all GSTs waiting for submission to this DBMS are retested (see Line (6)). The condition for submission is nearly the same as when the GT initially arrived. The candidate set is formed at line (7), but it is necessary to remove the GT which we are attempting to submit the subtransaction for and any which are inactive. The balance of the algorithm is conceptually identical to the original arrival.

Example 4.10 uses the same set and sequence of global transactions as in Example 4.9, which required the passivation of GT_3. In Example 4.10 it becomes apparent that the AGSS is more aggressive in that, partial GTs are submitted whenever it is safe.

Example 4.10 The Aggressive Global Serial Scheduler (AGSS) algorithm is demonstrated by the following:

1. GT_1 is presented to the AGSS and the required sets are formed. Since this GT only has one subtransaction (GST_1^1), it can be submitted. The Active_set($DBMS^1$) is updated and a record of the GSTs active status at $DBMS^1$ is made in the GSTs_active(GT_1) set. Line (1) of Figure 4.10 depicts the situation.

2. GT_2 arrives and since it accesses two DBMSs the initial **if** (line (6)) fails so each GST must be tested individually in the **for each** loop (line (11)). First,

Algorithm 4.2 *(Aggressive Global Serial Scheduler - Initial Scheduler)*

```
begin
  input   $GT_i$ : global transaction;
  var     candidate_set : set of GT identifiers;
  $GST_i \leftarrow$ makesubs($GT_i$);
  DBMS_set($GT_i$) $\leftarrow$ set of DBMSs accessed;
  GSTs_active($GT_i$) $\leftarrow \phi$;
  GSTs_passive($GT_i$) $\leftarrow \phi$;
  GSTs_complete($GT_i$) $\leftarrow \phi$;
  if card (DBMS_set ($GT_i$)) = 1 then
    begin
      k $\leftarrow$ database accessed;
      Active_set($DBMS^k$) $\leftarrow$ Active_set($DBMS^k$) $\cup GT_i$;
      GSTs_active($GT_i$) $\leftarrow$ k;
      submit $GST_i^k$ to $DBMS^k$
    end
  else
    for each $GST_i^k \in$ DBMS_set($GT_i$) do
      begin
        candidate_set $\leftarrow$ Active_set($DBMS^k$) $\bigcup$ Wait_Q($DBMS^k$) $- GT_i$
        for each $GT_m \in$ candidate_set do
          if $\exists$ active $GT_n$ such that
              DBMS_set($GT_m$) $\cap$ DBMS_set($GT_n$) $\neq \phi$ then
            candidate_set $\leftarrow$ candidate_set $\cup GT_n$
          endif
        endfor
        candidate_set $\leftarrow \{GT_k \mid GT_k \in$ candidate_set $\wedge$ card $(GST_k) > 1\}$
        if $\exists GT_j \in$ candidate_set $(m \neq n)$ such that
            ((GSTs_active($GT_j$) $\cup$ GSTs_passive($GT_j$)) $\cap$
            (DBMS_set($GT_i$) $- DBMS^k$) $\neq \phi$) then
          begin
            GSTs_passive($GT_i$) $\leftarrow$ GSTs_passive($GT_i$) $\cup DBMS^k$;
            passivate $GST_i^k$ on Wait_Q($DBMS^k$);
          end
        else begin
            GSTs_active($GT_i$) $\leftarrow$ GSTs_active($GT_i$) $\cup DBMS^k$;
            Active_set($DBMS^k$) $\leftarrow$ Active_set($DBMS^k$) $\cup GT_i$;
            submit $GST_i^k$ to $DBMS^k$;
          end
        endif
      end
    endfor
  endif
end
```

Figure 4.8: Aggressive Global Serial Scheduler (Part 1)

Algorithm 4.2 *(Aggressive Global Serial Scheduler - Subtransaction Termination)*

```
begin
  input   $GST_i^k$ : GST for $GT_i$ at $DBMS^k$ completes;

  var     candidate_set : set of GT identifiers;

  Active_set($DBMS^k$) ← Active_set($DBMS^k$) − $GT_i$;                                  (1)
  GSTs_active($GT_i$) ← GSTs_active($GT_i$) − $DBMS^k$;                                   (2)
  GSTs_complete($GT_i$) ← GSTs_complete($GT_i$) ∪ $DBMS^k$;                               (3)
  if GSTs_complete($GT_i$) = DBMS_set($GT_i$) then                                         (4)
    begin
      transaction complete;                                                                (5)
      for each $GST_j^k \in$ Wait_Q($DBMS^k$) do                                           (6)
        begin
          candidate_set ← Active_set($DBMS^k$) ∪
                ({$GT_l$|$GT_l \in$ Wait_Q($DBMS^k$) ∧ $GT_l$ is active}) − $GT_j$);       (7)
          for each $GT_m \in$ candidate_set do                                             (8)
            if ∃ active $GT_n$ ($m \neq n$) such that
                DBMS_set($GT_m$) ∩ DBMS_set($GT_n$) $\neq \phi$ then                       (9)
              candidate_set ← candidate_set ∪ $GT_n$                                      (10)
            endif
          endfor
          candidate_set ← {$GT_k$ | $GT_k \in$ candidate_set ∧ card ($GST_k$) > 1}        (11)
          if ∃$GT_m \in$ candidate_set such that
                ((GSTs_active($GT_m$) ∪ GSTs_passive($GT_m$)) ∩
                (DBMS_set($GT_j$) - $DBMS^k$) $\neq \phi$) then                            (12)
            repassivate $GST_j^k$ on Wait_Q ($DBMS^k$);                                   (13)
          else begin
                GSTs_passive($GT_j$) ← GSTs_passive($GT_j$) − $DBMS^k$;                   (14)
                GSTs_active($GT_i$) ← GSTs_active($GT_j$) ∪ $DBMS^k$;                     (15)
                Active_set($DBMS^k$) ← Active_set($DBMS^k$) ∪ $GT_j$;                     (16)
                remove $GST_j^k$ from Wait_Q($DBMS^k$);                                   (17)
                submit $GST_j^k$ to $DBMS^k$;                                             (18)
            end
          endif
        end
      endfor
    end
  endif
end
```

Figure 4.9: Aggressive Global Serial Scheduler (Part 2)

the AGSS attempts to submit GST_2^1 to $DBMS^1$. The only active transaction is GT_1, which is removed at line (16) because it only submits to a single DBMS. Since the **if** (line (17)) fails, the **else** portion is executed which submits the GST (line (20-22)). The set GSTs_active(GT_2) = $\{DBMS^1\}$ and the Active_set is updated (see line (2) of Figure 4.10). Secondly, GST_1^2 is tested and since Active_set($DBMS^2$) = ϕ, no intersection is possible, so the **if** (line(17)) fails and the subtransaction is submitted. The GSTs_active(GT_2) set is $\{DBMS^1, DBMS^2\}$.

3. GT_3 requires the submission of three GSTs. Each must be tested as above. The set of possible candidates is $\{GT_1, GT_2\}$ (see line (12-15)). (Since GT_1 will not effect the outcome of the test it is removed from the candidate_set at line (16). Therefore, it will be ignored in this and subsequent iterations.) Consider the possible submission of GST_3^1 to $DBMS^1$:

test for GT_2 at $DBMS^1$

$$\begin{aligned} &= (\text{GSTs\_active}(GT_2) \cup \text{GSTs\_passive}(GT_2)) \cap \\ &\quad (\text{DBMS\_set}(GT_3) - DBMS^k) \\ &= (\{DBMS^1, DBMS^2\} \cup \phi) \cap \\ &\quad (\{DBMS^1, DBMS^2, DBMS^3\} - DBMS^1) \\ &= \{DBMS^2\} \end{aligned}$$

This means that an overlap exists with active transactions at $DBMS^1$ so GST_3^1 must be placed on the Wait_Q($DBMS^1$) and GSTs_passive(GT_3) = $\{DBMS^1\}$. The following test demonstrates that GST_3^2 must be placed on Wait_Q($DBMS^2$).

test for GT_2 at $DBMS^2$

$$\begin{aligned} &= (\text{GSTs\_active}(GT_2) \cup \text{GSTs\_passive}(GT_2)) \cap \\ &\quad (\text{DBMS\_set}(GT_3) - DBMS^k) \end{aligned}$$

$$\begin{aligned} &= (\{DBMS^1, DBMS^2\} \cup \phi) \cap \\ &\quad (\{DBMS^1, DBMS^2, DBMS^3\} - DBMS^2) \\ &= \{DBMS^1\} \end{aligned}$$

Since there are no GSTs currently in the Active_set($DBMS^3$) or in its Wait_Q, the test fails and the **else** (line (20–22)) is executed leaving GSTs_passive(GT_3) = $\{DBMS^1, DBMS^2\}$. The Active_set and GSTs_active sets are updated and GST_3^3 is submitted to $DBMS^3$ (see line (3) of Figure 4.10).

4. GT_4 must undergo the same testing procedure. The candidates for testing are $\{GT_2, GT_3\}$ formed at line (12–16). The condition is existential so only one needs to occur for the condition to succeed:

test for GT_2 at $DBMS^2$

$$\begin{aligned} &= (\text{GSTs\_active}(GT_2) \cup \text{GSTs\_passive}(GT_2)) \cap \\ &\quad (\text{DBMS\_set}(GT_4) - DBMS^k) \\ &= (\{DBMS^2\} \cup \phi) \cap \\ &\quad (\{DBMS^2, DBMS^3\} - DBMS^2) \\ &= \phi \end{aligned}$$

test for GT_3 at $DBMS^2$

$$\begin{aligned} &= (\text{GSTs\_active}(GT_3) \cup \text{GSTs\_passive}(GT_3)) \cap \\ &\quad (\text{DBMS\_set}(GT_4) - DBMS^k) \\ &= (\{DBMS^3\} \cup \{DBMS^1, DBMS^2\}) \cap \\ &\quad (\{DBMS^2, DBMS^3\} - DBMS^2) \\ &= \{DBMS^3\} \end{aligned}$$

Since the test succeeds in the second case, the transaction must be passivated to ensure correct MDB-serializability. The second GST must be tested. The candidate set is formed $\{GT_2, GT_3\}$, at line (12–16). Consider the test:

test for GT_3 at $DBMS^3$

$$\begin{aligned}
&= (\text{GSTs\_active}(GT_3) \cup \text{GSTs\_passive}(GT_3)) \cap \\
&\quad (\text{DBMS\_set}(GT_4) - DBMS^k) \\
&= (\{DBMS^3\} \cup \{DBMS^1, DBMS^2\}) \cap \\
&\quad (\{DBMS^2, DBMS^3\} - DBMS^3) \\
&= \{DBMS^2\}
\end{aligned}$$

Therefore, this GST must be passivated and the appropriate sets updated. Since no subtransactions were submitted the Active_sets remain unchanged leaving GSTs_passive(GT_4) = $\{DBMS^2, DBMS^3\}$.

5. GT_5 is submitted immediately, see line (5) of Figure 4.10.

Transactions terminate in their submission same order.

1. GST_1^1 completes so GT_1 is removed from the Active_set($DBMS^1$). The GST is taken from GT_1's active set and placed in its GSTs_complete set. Since this is the only GST in the DBMS_set(GT_1), the transaction terminates and the Wait_Q($DBMS^1$) is signaled so waiting GSTs may begin. The subtransaction completion process tries to schedule any GSTs in the Wait_Q at $DBMS^1$. The only candidate is GT_3's GST_3^1 but the condition fails because GT_2 is active at $DBMS^1$ and $DBMS^2$ (line (6) of Figure 4.10).

2. GST_2^1 completes so GT_2 is removed from the Active_set($DBMS^1$). $DBMS^1$ is removed from GSTs_active(GT_2) set and included in GSTs_complete(GT_2) set. Since all GSTs are not complete the **transaction complete** is not executed. When GST_2^2 terminates, GT_2 is completed so GST_3^1 and GST_3^2 are tested for submission. Recall that the GSTs waiting for submission at $DBMS^1$ and $DBMS^2$ are $GST_3^1, GST_3^2, \text{and} GST_4^2$. Each is tested for submission at line

	Active_set($DBMS^1$)	Active_set($DBMS^2$)	Active_set($DBMS^3$)
(0)	ϕ	ϕ	ϕ
(1)	$\{GT_1\}$	ϕ	ϕ
(2)	$\{GT_1, GT_2\}$	$\{GT_2\}$	ϕ
(3)	$\{GT_1, GT_2\}$	$\{GT_2\}$	$\{GT_3\}$
(4)	$\{GT_1, GT_2\}$	$\{GT_2\}$	$\{GT_3\}$
(5)	$\{GT_1, GT_2\}$	$\{GT_2\}$	$\{GT_3, GT_5\}$
(6)	$\{GT_2\}$	$\{GT_2\}$	$\{GT_3, GT_5\}$
(7)	$\{GT_3\}$	$\{GT_2\}$	$\{GT_3, GT_5\}$
(8)	$\{GT_3\}$	$\{GT_3\}$	$\{GT_3, GT_5\}$
(9)	$\{GT_3\}$	$\{GT_3\}$	$\{GT_4, GT_5\}$

Figure 4.10: Trace of Scheduler Execution for Example 4.4

(6). Consider the attempt to submit GST_3^1. Since no GTs are currently active at $DBMS^2$ the candidate_set $= \phi$ at line (7-11). No GT exists which prevents its submission, so line (14-18) submits it. GST_3^2 is submitted in the same way.

Consider the possible submission of GST_4^2. The candidate_set $\{GT_3\}$ is formed at line (7-11). The **if** test (line (12)) is as follows: $(\{DBMS^1, DBMS^2, DBMS^3\} \cup \phi) \cap (\{DBMS^2, DBMS^3\} - DBMS^2) = \{DBMS^1, DBMS^2, DBMS^3\} \cap \{DBMS^3\} = \{DBMS^3\} \neq \phi$ so GST_4^2 must be **repassivated** (line (13)).

3. GST_3^3 terminates. The process is the same as above so GST_4^2 and GST_4^3 are tested for submissability. The only transaction which could cause the passivation of this subtransaction is GT_5 but the test fails vacuo[illegible]. (See line (9) of Figure 4.10).

4. The balance of the GSTs termination in an obvious way. ◇

Example 4.10 demonstrates the submission of part of GT_3 in an aggressive way, which would have been blocked entirely by the GSS algorithm. Unfortunately, the AGSS's decision to submit a GST from GT_3 causes GT_4 to be passivated. The best algorithm depends upon the MDMS environment and is the subject of future research.

4.4.4 Correctness of the Aggressive Global Serial Scheduler

Proving the correctness of the AGSS requires the same two step process used for the GSS. We must classify the set of histories produced by the scheduler and demonstrate that all such histories are MDB-serializable or equivalently, that every such history produces only acyclic MSGs. The only step which differs from the earlier proof is in the classification of the histories. Recall Lemma 4.2 stated that all global subtransactions at every DBMS are scheduled in the same way. Since Proposition 4.1 holds, it is only necessary to prove that Lemma 4.2 is true and the proof follows by appealing to Theorem 4.3. Therefore, it is sufficient to prove the following Lemma.

Lemma 4.3 For every pair of global transactions $GT_i, GT_j \in \mathcal{GT}$ scheduled by the AGSS either all of GT_i's subtransactions are executed before GT_j's at every DBMS or vice versa.

Proof: All global transactions requiring only a single subtransaction (eg. $GST_i^k \in GT_i$) are always scheduled immediately (line (6) Figure 4.8). From Proposition 4.1 it follows that any other $GST_j^k \in GT_j$ either precedes or follows GST_i^k.

Global transactions which submit to multiple database but only access one database in common with all other active global transactions are submitted immediately. This occurs because each GST is tested independently (line (11) Figure 4.8). Since only one DBMS is accessed in common for each GT in the candidate set (line (12-16)), the test will fail at (line (17)) of Figure 4.8) which results the submission of each and every GST. Since each GT in the candidate_set overlaps at only one DBMS, it follows without loss of generality that, when we consider a $GST_i^k \in GT_i$ and $GST_j^k \in GT_j$ whose base-set overlap, any ordering chosen by $DBMS^k$ is consistent. That is, since

$DBMS^k$ is the only place where GT_i and GT_j access common data, we know from Proposition 4.1 that either $GT_i \prec GT_j$ or $GT_j \prec GT_i$, as required.

Global transactions that overlap at more than a single DBMS are always submitted and executed in the same order at each overlapping DBMS. Consider two arbitrary active global subtransactions $GST_j^k, GST_j^l \in GT_j$ which access $DBMS^k$ and $DBMS^l$, respectively. Consider the attempt to submit GST_i^k. Since there exists another GST at $DBMS^l$, either waiting or active, GST_i^k is passivated (line (18–19) of Figure 4.8 blocks this GST). GST_i^l is blocked for the same reason. Consider the completion of GT_j. Assuming that there are no other similar GSTs, the test at line (12) of Figure 4.9 permits GST_i^k to be submitted. Since GST_j^k is complete, it follows immediately that $GST_j^k \prec GST_i^k$. GST_i^l is submitted in the same way. It follows that $GST_j^l \prec GST_i^l$. Clearly the GSTs of GT_j precede the GSTs of GT_i at every DBMS, as required. □

4.5 Concluding Remarks

Controlling the execution order of global subtransactions at autonomous DBMSs is accomplished by embedding ordering information in the subtransactions or by controlling their submission. Georgakopoulos and Rusinkiewicz's forced local conflicts and Litwin and Tirri's value date methods are examples of approaches which use embedded information to order subtransaction execution. However, more closely related to the GSS and AGSS algorithms are the altruistic locking [1, 38] and site graph approaches [6]. These approaches control the effective execution order of global subtransactions by controlling their submission.

Altruistic locking is based on the concept of a transaction's *wake*. The wake of a transaction consists of all the data items that it has locked and accessed but will not access again. The fundamental idea is that a transaction can be safely permitted to lock (and access) a data item (without having to wait for the release of previous locks on it) if it remains completely in or completely out of the wakes of all transactions that currently hold locks on it. Thus, in its most basic implementation, altruistic locking would schedule transactions to access locked data items as long as they are completely in the wake of all the transactions that hold locks on those data items. It is possible to relax this strict implementation by defining conditions where a transaction can "straddle" the wake (i.e., be partially in the wake and partially outside the wake) of another transaction and still be permitted to access the locked data item safely. This is done by declaring the data items that a transaction is later going to lock by means of a "mark" operation. A more precise definition of these conditions is given in Salem *et al.* [38].

Altruistic locking has been applied to the MDB environment by assuming an execution model similar to ours [1]. The major difference is that each of the individual databases is considered a synchronization unit. Therefore, individual global transactions lock each local database before submitting subtransactions. Any subsequent global transaction which stay completely within the wake of a transaction which already holds a lock on a local database is allowed to submit a subtransaction to the associated DBMS. The concept of marking is also extended to operate on granules of individual databases.

The difference between our schedulers and Salem *et al.*'s implementation of altruistic locking is the additional concurrency provided by the GSS and AGSS. Since each global transaction locks each DBMS, with altruistic locking, only one subtransaction can be submitted to a DBMS at any given time. This is equivalent to executing global transactions serially at each individual DBMS. Both the GSS and AGSS permit multiple global subtransactions to execute at a DBMS at the same time. The conditions that are defined in this chapter allow for this added concurrency and contain the wake definition of altruistic locking mechanism.

Site graphs uses the same all or nothing approach to subtransaction submission adopted by the GSS. The approach is to form site graphs based on the data items accessed and their location. If a cycle free site graph is formed the global transaction's subtransactions can be submitted to the DBMSs for execution. The AGSS differs from the site graph algorithm by attempting to submit any subtransaction which can be safely executed.

Chapter 5

MDS Reliability

Reliability is comprised of two parts. First, a system is reliable if the effects of committed transactions are reflected on the database, but the effects of uncommitted or aborted transaction do not appear, that is, it assures transaction atomicity. Second, reliability requires that in the event of a system failure database is recovered to a consistent state so that transactions terminate according to the first condition. This is known as crash recovery.

This chapter is presented as follows. First, the types of failures possible in a MDS, given the computational model defined in Chapter 3, are stated (Section 5.1). The architecture of the MDMS's recovery manager is presented next (Section 5.2). Since the first part of the reliability is transaction atomicity, a two-phase commit algorithm which ensures DBMS autonomy is then described (Section 5.3). Crash recovery is the second part of reliability, so we state what is required of each DBMS to guarantee proper recovery (Section 5.4). Finally, the correctness of the transaction atomicity and crash recovery algorithms are demonstrated (Section 5.5 and Section 5.6, respectively).

5.1 Types of Failures

Failures in a multidatabase systems can occur in different ways:

1. DBMS (MDMS) failure: happen when a condition occurs which stops the execution of a DBMS (MDMS). The failure of a DBMS does not cause a total system failure since other DBMSs may continue to function.
2. Transaction failure: occurs for two possible reasons.
   - (a) The transaction may abort itself, which occurs most often because a value dependency test fails. These are called *value dependency failures*.
   - (b) A DBMS may abort a transaction because of some external event, such as a deadlock.
3. System failure: occurs when the underlying operating system suffers a failure which stops the DBMS. The operating system may be partially operational, but the DBMS has failed because of some malfunction. This has the same affect as a DBMS failure except the operating system requires maintenance. This is important since our environment is composed of a single site and it is possible to have one DBMS fail but others remain operational.
4. Media failure: occurs when data is lost from secondary storage.

System failures are beyond the scope of this thesis. When a DBMS fails as a result of the failure of an operating system; the system must be restored before the DBMS can become operational. This research deals with failures of the DBMS (MDMS), so the reasons for the failure are independent of their management. We assume, throughout the balance of this thesis, that the operating system is functioning correctly before the DBMS attempts to become operational. Since the concern is for DBMS failures, we will refer to system failures as DBMS failures even if the "fault" is not with the DBMS *per se*.

DBMS and transaction failures are discussed in Section 5.1.1 and Section 5.1.2, respectively. Media failures are discussed in Section 5.1.3.

5.1.1 DBMS Failure

Recovery from DBMS failures is based on the information stored on secondary, nonvolatile storage. The information stored on secondary storage is referred to as the log of transaction execution or simply as the *log*. A transaction's actions must be logged when it reads or writes a data item, or when it begins or terminates execution. The most critical information logged is the commitment decision. A transaction is not considered committed until the commitment is logged by the DBMS. The information necessary to ensure that the effects of all committed transactions are reflected in the database is obtained from the log at recovery time.[1]

Two techniques are available for DBMS recovery along with some hybrids of these techniques [3]. The first approach is to **redo** all operations belonging to committed transactions and whose actions were not recorded in the database at the time of the failure. The second option is to **undo** operations affected the database but whose transaction was not committed. Recovery management algorithms have been defined for both forms of recovery. Specific details concerning undo versus redo are covered in Bernstein *et al.* [3] or Özsu and Valduriez [34], so they are not discussed in detail here.

Database system failures in a MDS can be due to software or hardware errors. Software failures can occur in three possible ways. First, one or more of the participating DBMSs can fail. Second, the MDMS can fail. Third, the MDMS and one or more of the DBMSs can fail simultaneously.

An assumption resulting from the autonomy of the DBMSs is that each is capable of managing all submitted transactions independently. It follows that when a DBMS fails, it is capable of recovering so only committed transactions affect the database. Further, once recovered, the DBMSs must be able to notify the transaction's submitter of the termination condition. Multiple DBMS failure are recovered from independently so the occurrence of multiple failures is no more complicated than the failure of a single DBMS.

---

[1] Detailed description of logging activities is delayed until the data manager's architecture is presented in Section 5.2.

Failure of the MDMS is different than that of a DBMS. Recovery is required so the effects of committed global transactions are reflected in each DBMS. The MDMS recovers suing global subtransactions since it does not manage data directly. Details are described in subsequent sections.

System failures of the MDMS and some of the DBMSs requires that all failed systems be restored. Since all systems are autonomous they can all be recovered independently. Once the MDMS resumes operation it must determine which DBMSs are functioning and ensure that only the effects of committed transactions are reflected. Failure of the MDMS and some DBMSs is no more complicated than the failure of either. The reasons for this will become apparent in Section 5.3.

5.1.2 Transaction Failure

Transactions fail in two ways: a value dependency test fails or a system requires the failure of the transaction for some other reason (e.g. resolving deadlocks). Transaction failures result in aborting the transaction and undoing its effects. Both types of transaction failures can occur at either the DBMS or MDMS levels, which are called local and global, respectively.

Consider the two possible reasons for failure at the local level. Since each DBMS is autonomous and correct, it follows that the DBMS can correctly recover from transaction failures at the local level. Furthermore, since the aborting of a transaction is not necessarily self-induced it is impossible to assume that global subtransaction only abort as a result of a value-dependency failure.

Failure of global transactions can also occur as a result of a value dependency check. The problem is difficult because the value dependency is detected by a global subtransaction, not by the global transaction. Consider how a value dependency failure is detected by the MDMS. A GST is submitted to a DBMS where the incorrect value dependency is detected, which causes the GST to fail. The failed GST is reported to the MDMS, which must take appropriate actions to either correct the value dependency or fail the global transaction. Failing the GT requires the failure of all global subtransactions submitted on its behalf. GTs may also be aborted by an external event. This could occur when the MDMS signals a failure or a single DBMS aborts a global transaction's GST.

5.1.3 Media Failures

Since each DBMS is autonomous and function correctly, recovery from local media failures a local responsibility. Although the loss of data from a media failure could effect the MDMS it will do so in the same way as any other user. When a GST commits on behalf of a global transaction the DBMS is responsible for ensuring that its effects are reflected in the database. We assumes that media failures are recovered from without user interaction. Since the MDMS is like any other user and a DBMS can recover from local media failures without user assistance, the global level is not required for recovery from media failures. Since the MDMS must manage logs as well, the global storage must be backed up in the face of media failures. Processes for this are well-understood in a centralized DBMS environment and could be readily adapted to the MDMS.

5.2 MDMS Data Management Architecture

The data management architecture is discussed in a bottom-up fashion by first considering local data managers and then presenting the global data management functions that need to be implemented on top of the local one. Finally, the contents of the global log and the information maintained by the Global Recovery Manager is presented.

5.2.1 Local Data Manager Architecture

The interface between the local scheduler and the local data manager consists of transaction operations such as read, write, commit, and abort. These are submitted to the local data manager by the local scheduler which performs the necessary operations to execute the transaction. Once an operation is serviced the result is returned to the scheduler, including an acknowledgment of its success or failure. Figure 5.1 depicts this architecture and is similar to the model in Bernstein *et al.* [3], but has been modified to emphasize the focus of this research.

A local data manager has a local cache and a local stable storage. These resources are managed by the local recovery manager and the local cache manager. The local cache manager and the local recovery manager can read and write volatile memory (local cache) with operating system level reads and writes.[2] If a data item is not available in the local cache for the local recovery manager, a fetch is issued to the local cache manager to retrieve the necessary item from database. A paging algorithm is used to determine how the page is placed in local cache. These algorithms are well known [12] and their description is beyond the scope of this paper. If a specific data item must be stored in stable storage the local recovery manager issues a flush operation to the local cache manager which causes it to be written to

---

[2] The read and write from/to LC and from/to LSS are different than the read and write issued by transactions. The former are low level operating system reads that actually touch a physical memory location while the latter are logical reads and writes to a logical data item. When we discuss reading and writing the type should be obvious from the context.

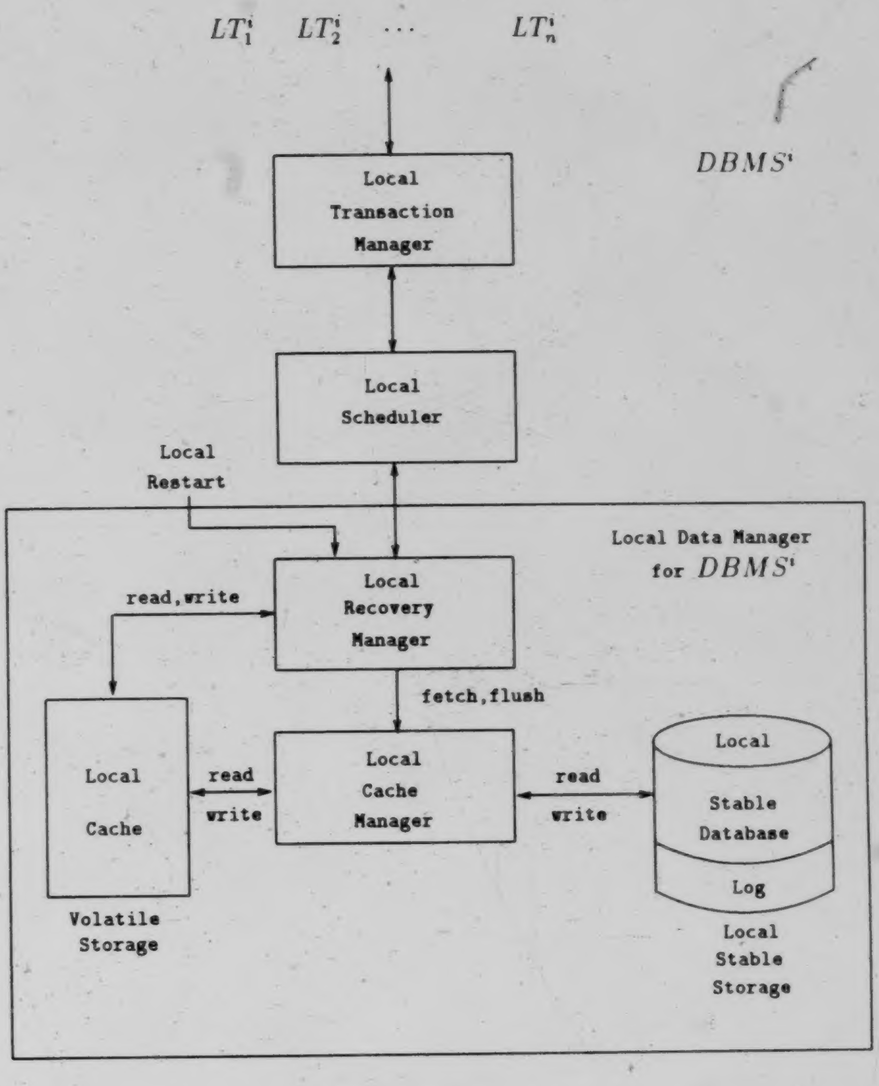

LT_1^i LT_2^i ... LT_n^i
DBMS^i
Local Transaction Manager
Local Scheduler
Local Restart
Local Data Manager for DBMS^i
read,write
Local Recovery Manager
fetch,flush
Local Cache
read write
Local Cache Manager
read write
Local Stable Database
Log
Volatile Storage
Local Stable Storage

Figure 5.1: Generic Model of a Local Data Manager

the local stable storage.

In the event of a transaction failure the local recovery manager is responsible for ensuring that only the effects of committed transactions are reflected in the database. Techniques for this are well understood so they are not discussed here. In the event of a DBMS failure the local recovery manager must be able to restore the database using information maintained in the local log. This is accomplished by issuing a local restart operation after the DBMS is repaired but before the local recovery manager accepts any operations from the local scheduler. Local restart is considered idempotent so that if the DBMS fails during the restart, the operation can be rerun.

A number of steps are required when a DBMS fails that it can resume normal operations. Local restart typically involves the following sequence of steps which could be performed interactively by a database administrator or as a batch process:

Local Restart Process

1. Start the DBMS as a process on the operating system.
2. Recover the database by using information kept in the local stable storage log.
3. Open the DBMS for user access, that is, permit transactions to execute.
4. Terminate the local restart process.

During the restart process only the database administrator is permitted to access the database until after the recovery has completed at Step 2. Note that the opening of the DBMS to user access in Step 3 does not require that the database administrator permit access to all possible users.

5.2.2 Global Recovery Manager Architecture

Unlike a centralized DBMS, which manages the operations of transactions, the global recovery manager manages sets of operations in the form of global subtransactions. The MDMS does not directly manipulate any data so the global recovery manager does not manage reads and writes to the databases. The global recovery manager must ensure that critical information is placed on stable storage so recovery is possible from any type of failure. The global recovery manager must ensure that, in the event of a failure, all DBMSs retain only the effects of committed global transactions. If the it guarantees this and each DBMS is reliable, then the MDS can recover from any type of failure.

Figure 5.2 illustrates the architecture of the global recovery manager in relation to the global transaction manager and the global scheduler. The global execution monitor receives GSTs which must be submitted to the DBMS where is will be serviced. Once a GST completes, the termination condition is returned to the global scheduler.

The global recovery manager requires a global stable storage. In the event of a system failure, all information in volatile memory is lost so information which must be saved is placed in global stable storage.

Finally, if the MDMS fails a global restart operation is issued which will reestablish the status of outstanding global transactions and global subtransactions. Since the MDMS does not manipulate data, the task of the global restart is to ensure that all global transaction's GSTs complete correctly.

Critical information required in the event of a failure must be written directly to the global stable storage, specifically the global log. The purpose of this research is to ensure that a failed MDMS can recover using a global restart operation. The architecture will be expanded to emphasize the role of the global log.

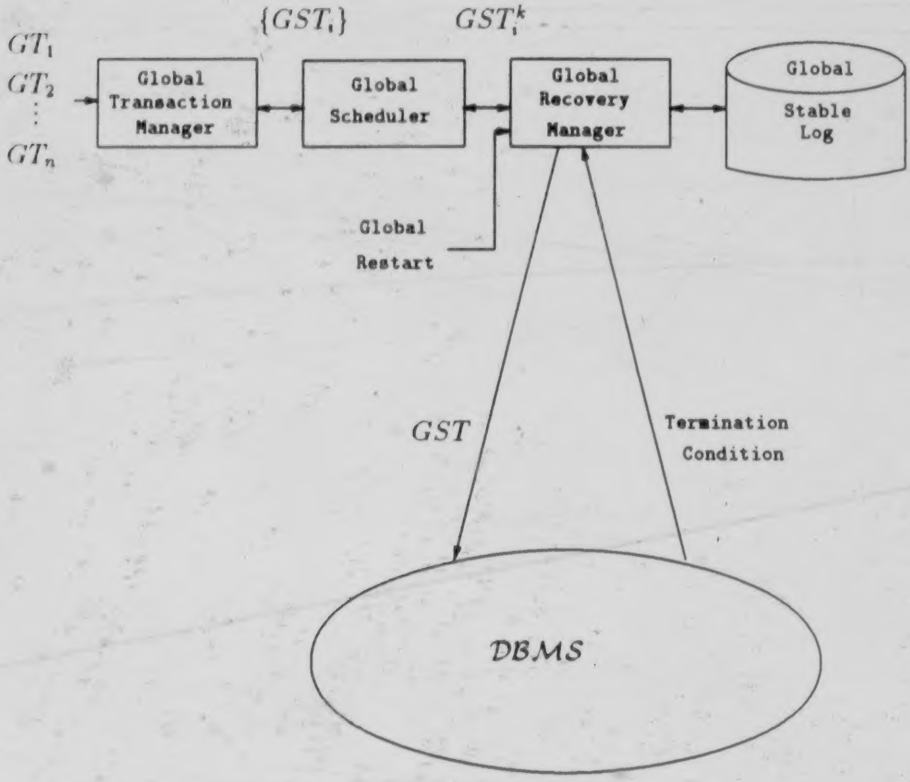

GT_1
GT_2
GT_n
$\{GST_i\}$
GST_i^k
Global Transaction Manager
Global Scheduler
Global Recovery Manager
Global Stable Log
Global Restart
GST
Termination Condition
$\mathcal{DBMS}$

Figure 5.2: Architecture of the Global Recovery Manager

5.2.3 Global Logging

Figure 5.3 expands the architecture of Figure 5.2 with emphasis on the log. Critical stages in a global subtransaction's execution must be logged to ensure consistency in the presence of failures. The global scheduler transmits three types of information to the global recovery manager, namely, global transaction initiation messages, global subtransactions, and the global transaction's termination condition.

1. Before submitting GSTs the global scheduler informs the global recovery manager of new global transactions by passing a begin global transactions (B_GT) operation composed of two parts:
    (a) A global transaction identifier.
    (b) A list of global subtransaction identifiers to be processed for this global transaction.

    An explicit end global transaction is not necessary because a global transaction is considered complete once all GSTs have been processed and the global scheduler transmits a termination operation. A GT terminates when all of its GSTs have committed or the global scheduler transmits an abort operation to terminate further processing. However, it may be useful to write an explicit end global transaction in the global log to aid in the recovery process.

2. The global subtransaction is sent to the global recovery manager which logs it before submission to the required DBMS. The global recovery manager will respond to global scheduler with the termination condition received from the DBMS.

3. The global recovery manager must record each GT's termination condition to ensure that only committed global transactions affect the DBMSs. The global recovery manager will informs the global scheduler of the termination condition, at which time the GT is committed.

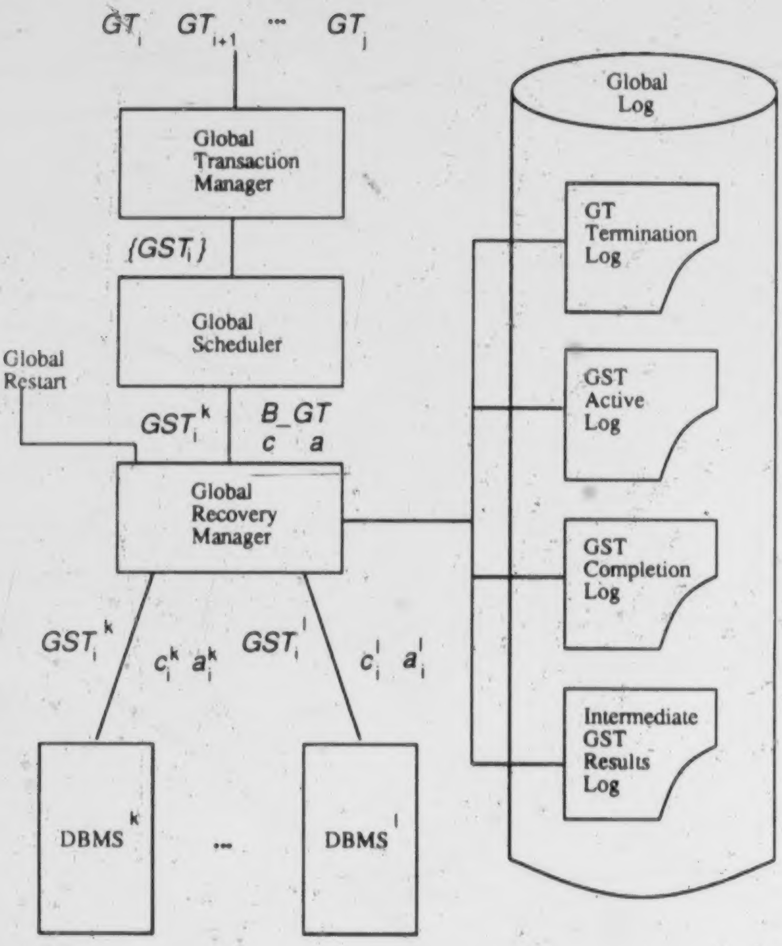

GT_i GT_{i+1} ... GT_j
Global Transaction Manager
$\{GST_i\}$
Global Scheduler
Global Restart
GST_i^k
B_GT
c a
Global Recovery Manager
GST_i^k
c_i^k a_i^k
GST_i^l
c_i^l a_i^l
DBMS^k
...
DBMS^l
Global Log
GT Termination Log
GST Active Log
GST Completion Log
Intermediate GST Results Log

Figure 5.3: Detailed Architecture of the Global Recovery Manager

Although a single global log is kept on stable storage, four critical points must be logged. First, the termination condition of all global transactions must be recorded even if the decision is to abort. This is required in the event of a MDMS failure and the reason will be discussed in subsequent sections. Secondly, each GST which has completed is recorded and the termination condition is logged. Thirdly, when a GST has been submitted to a DBMS it is recorded on an active list in the log so the global restart operation knows which GSTs may still be outstanding. Finally, when a GST completes, information may be returned in the form of a result. This information must be save so that it is available in the event of a failure while waiting for outstanding GSTs.

5.3 Two phase Commit in MDSs

In this section the well known two-phase commit approach used in distributed databases is extended to the MDB environment. The protocol operates differently in the MDB environment. Distributed database systems are typically tightly integrated so the participating data managers communicate, enabling synchronization of actions. MDSs ~~do not~~ enjoy the luxury of inter-DBMS communication and synchronization. Furthermore, distributed database recovery managers which use a two-phase commit have a special state known as READY or PRE-COMMIT (see Özsu and Valdriuz [34]). When a transaction enters the state, the desired termination decision is logged, so that, if a DBMS fails information required for recovery will not be lost. However, the autonomous DBMSs implement a one-phase commit where they execute a commit operation by logging the transaction using the well known write-ahead logging protocol [24]. Since the READY state does not exist explicitly, a technique to simulate it is required.

5.3.1 Overview

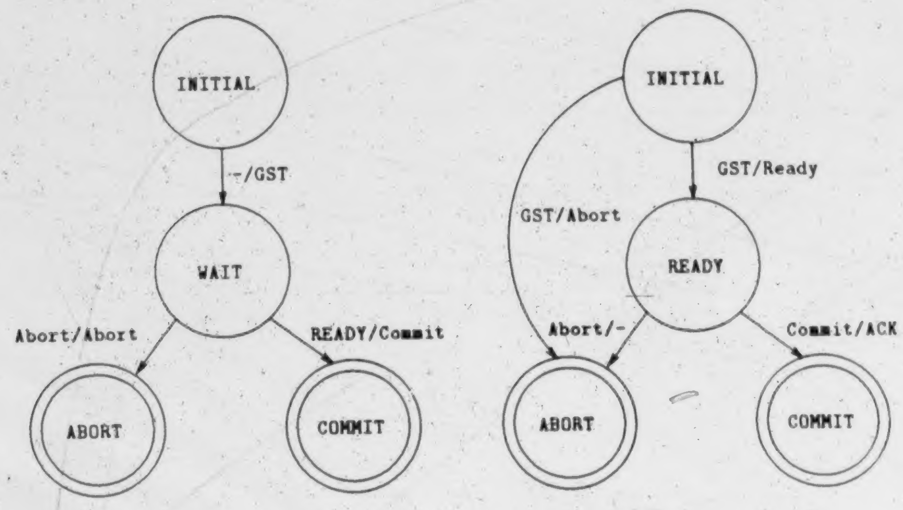
Figure 5.4 depicts the state diagram of the two-phase commit protocol. The challenge is to implement these state transitions on a multidatabase system where communication between the DBMSs is not possible. The manner in which the decisions are made is different in a multidatabase system than a distributed database system.

The MDMS prepares the global transactions for submission to the DBMSs (INITIAL state). Once all GSTs have been submitted to their respective DBMSs, the GT is moved to the WAIT state. If all of the GSTs become ready the GT is moved to the COMMIT state. If any one of the GSTs do not become ready the GT moves to the ABORT state. Once a GST is submitted to a DBMS it is in the INITIAL state. The GST remains in this state until it decides to abort unilaterally whereby the GST moves to the ABORT state, or it is ready to commit, in which case it moves to the READY state. Once the GST is in the READY state it waits for the final commit decision from the MDMS. If the GT commits, a message is sent to the local level so the GST can move to the COMMIT state. If the GT is aborted, the GST moves to the ABORT state.

The difficulty lies in maintaining the state transitions at the DBMSs. The problem is getting the individual DBMSs to behave according to the two-phase commit protocol when they only implement a single-phase. The rest of this chapter discusses implementation issue.

5.3.2 The Technique

Previous research identified the need for a two- phase commit solution to the MDS transaction management problem (see Elmagarmid and Helal [15] for example). The major difficulty of implementing two-phase commit is the autonomy of the DBMSs which cannot be modified. This section describes a technique that simulates two-phase commit without modifying the DBMSs. Consider the following generic global


INITIAL
-/GST
WAIT
Abort/Abort
READY/Commit
ABORT
COMMIT
INITIAL
GST/Ready
GST/Abort
READY
Abort/-
Commit/ACK
ABORT
COMMIT

GTs at MDMS GSTs at DBMSs

Figure 5.4: State Diagram for MDB Two Phase Commit Algorithms

subtransaction:

```
GST:        ⋮
        if not < cond > then
            abort
        else
            commit;
```

The GST involves a sequence of read and write operations. At some point (possibly at many points) a condition is tested to determine if the GST should abort. In the event that a GST wishes to abort it can do so and the effects are not particularly interesting but will be discussed in a later section. If all conditions are met throughout its execution the GST decides to commit. The key to the algorithm is to modify each commit operation in the global subtransactions as follows:

```
GST:        ⋮
        if not < cond > then
            abort
        else
```

```
send ready accept q_cond;
if q_cond is abort or send failed then
    abort
else
    commit;
```

The modification amounts to sending a message to the MDMS in the middle of a subtransaction and blocking until a message is returned. The required program has been referred to by other authors ([13] or [37]) as a STUB process.

We briefly digress to compare and contrast the STUB process utilized in this research with those of other researchers. Pu's [37] STUB process is created to submit a subtransaction to a local DBMS. The STUB process proposed here performs this task as well as the necessary communication with the MDMS to simulate the READY state.

The STUB process suggested by Elmagarmid [13] is a procedure which takes as input both a sequence of subtransactions and an ordering vector used in scheduling the subtransactions. "At an abstract level, a stub process is considered as a function which takes as input the global subtransaction and a linear order over them, and produces as output a submission policy."[13, Page 39] Our approach differs in that the global subtransaction submission is left to the global scheduler and ordering of operations at each DBMS is performed by the local schedulers. Our STUB process only helps to synchronize commit activities between the DBMS and the global recovery manager to simulate two-phase commit. This is a simpler solution which permits greater autonomy since no ordering is forced on the DBMS.

Example 5.1 The feasibility of the proposed modification is demonstrated by assuming that $DBMS^1$ runs SQL and the MDMS submits global subtransaction to $DBMS^1$ in the same language. Assume that a global transaction has been parsed and the following global subtransaction must be submitted to $DBMS^1$:

BEGIN TRANSACTION GST_1^1

```
        UPDATE table
        SET attrib = new value
        WHERE condition
    COMMIT TRANSACTION
```

The MDMS must take this global subtransaction and produce one capable of emulating a READY state, as follows:

```
BEGIN TRANSACTION $GST_1^1$
    UPDATE table
    SET attrib = new value
    WHERE condition

call function which informs the MDMS that the
GST is ready and wait for a response

if response is ABORT then
    ROLLBACK TRANSACTION
else
    COMMIT TRANSACTION
```

□

We have demonstrated the requirements at an abstract level and are now in a position to illustrate its feasibility. In the following example we assume that $DBMS^1$ runs SYBASE[3] and global subtransactions are submitted to it by executing a C program which calls the required DBMS functions.

Example 5.2 This example demonstrates the feasibility of the approach rather than provide a detailed description of all the implementation details. Since the example uses SYBASE and C, the reader must be familiar with SYBASE, C and the access commands to SYBASE.

```
#include stdio.h
#include sybfront.h
#include sybdb.h
```

---

[3] SYBASE is a registered trademark of Sybase Inc.

```
#define DATELEN 26
#define TYPELEN 2

#define TRUE 1
#define FALSE 0

#define COMMIT 'C'                  /* The GST is committing or committed */
#define PTC    'P'                  /* The GST is ready to commit */
#define ABORT  'A'                  /* The GST is aborting or aborted */

int err_handler ();
int msg_handler ();
char MDMS_call ();

main ()
{
    DBPROCESS    *dbproc;               /* Our connection with SQL Server */
    LOGINREC     *login;                /* Our login information */

    DBCHAR    crdate[DATELEN+1];
    DBINT     id;
    DBCHAR    name[MAXNAME+1]; /* MAXNAME is defined in "sysdb.h"
                                * as the maximum length for names
                                * of database objects, such as
                                * tables, columns and procedures. */

    DBCHAR type[TYPELEN+1];
    RETCODE result_code;

    int term_flag;       /* termination condition for subtransaction */
    char set_exit;       /* ultimate commitment decision */
    char response;       /* result from a PTC call to the MDMS */

            /* Initialize DB-Library */
    if (dbinit() == FAIL)
        exit (ERREXIT);

        /* Install the user-supplied error-handling and message-handling
        routines. They would be defined at the bottom of this
        source file.*/

    dberrhandle (err_handler);
    dbmsghandle (msg_handler);
```

```
        /* Get a LOGINREC structure and fill it with the necessary login
           information. */

login = dblogin ();
DBSETLPWD (login, "server_password");
DBSETLAPP (login, "example1");

        /* Get a DBPROCESS structure for communicating with SQL Server.
           A NULL servername defaults to the server specified by DSQUERY. */

dbproc = dbopen (login, (char *) NULL);

/* First start the transaction. */

dbcmd (dbproc, "BEGIN TRANSACTION gst1");

            /* Place a sequence of SQL statement for in the
            command buffer for submission */

dbcmd (dbproc, "UPDATE table");
dbcmd (dbproc, "SET attrib1 = new value");
dbcmd (dbproc, "WHERE condition");

            /* Now submit the body of the transaction for
            execution */
dbsqlexec (dbproc);

                /* Get the results */
                /* Note that all of the results must be properly
                acquired in this section */
term_flag = TRUE;
while ((result_code = dbresults(dbproc)) != NO_MORE_RESULTS) {
    if (result_code == FAIL)        /* Some value was not correctly updated */
        term_flag = FALSE;
}

                    /* if some value was not updated correctly
                    the subtransaction must be aborted */
if term_flag == FALSE {
    dbcmd (dbproc, "ROLLBACK TRANSACTION");
    dbsqlexec(dbproc));
    set_exit = ABORT;
}
```

```
    /* Subtransaction is ready to commit */

    /* Communicate with the MDMS to indicated Ready to Commit */

    response = MDMS_call (PTC,ID);

    /* MDMS aborting PTC subtransaction */
    if response == ABORT {
        dbcmd (dbproc, “ROLLBACK TRANSACTION”);
        dbsqlexec(dbproc);
        set_exit = ABORT;
    } /* MDMS committing PTC subtransaction */
    else {
        dbcmd (dbproc, “COMMIT TRANSACTION”);
        dbsqlexec(dbproc);
        set_exit = COMMIT;
    }

    dbexit ();
    exit (set_exit);
}                                                            □
```

The code in Example 5.2 would be generated by the MDMS, not the user. The global user would simply submit a SQL query which is parsed into a set of global subtransactions. Each global subtransaction is then converted to a program similar to the one illustrated. Techniques for decomposing a global transaction into a set of subtransaction is currently an open research problem and beyond the scope of this thesis. We are in a position to discuss what the MDB two-phase commit protocol requires of the DBMSs.

5.3.3 Individual DBMS Requirements

The first issue is the level of reliability provided by the DBMSs. Earlier it was claimed that each DBMS must guarantee a locally strict level of service. The reason for this is demonstrated by assuming that a weaker service level is sufficient and then illustrating any shortcomings. Since the READY state can be viewed as an operation, it will be included explicitly (where applicable) in the histories. This operation would not appear in the local DBMS log but is useful when reasoning about the MDB two-phase commit protocol. Note that only a termination operation can follow the ready to commit. We use pc_i^k to indicate that GST_i^k is ready (prepared) to commit (i.e., in the READY state).

The following history demonstrates that ALCA is not adequate. Assume that $DBMS^k$ produces a history where GST_i^k is a global subtransaction for GT_i, LT_j^k is a local transaction, and $LDB^k = \{x\}$:

$$LH^1 : r_i^k(x); w_i^k(x); pc_i^k; \hat{w}_j^k(x); \hat{c}_j^k; c_i^k;$$

Further we assume that, based on GST_i^k's pc_i^k, the MDMS commits GT_i. Note that LT_j^k does not have a $\hat{pc}_j^k$ since local transactions do not have a READY state. Therefore, GST_i^k must commit. In the history above, the commit decision is received by GST_i^k so it commits correctly. In this serialization it is apparent that $GST_i^k \prec_{LH} LT_j^k$ since the operations of GST_i^k occur before the operations of LT_j^k.

Consider the effects of a system failure at $DBMS^k$ after $\hat{c}_j^k$ but before c_i^k. This means that LT_j^k is logged, but GST_j^k is not because it is not a committed transaction. Recall the steps involved in a typical local restart operation described in Section 5.2.1. The DBMS is restarted, the database is recovered based on the log, and then users are given access to the DBMS. Since GT_i has committed, GST_i^k must commit. Even if the DBMS now grants immediate and exclusive access to the MDMS the following history would result:

$$LH^1 : \hat{w}_j^k(x); \hat{c}_j^k; \quad (1)$$

$$r_i^k(x); w_i^k(x); pc_i^k; c_i^k; \quad (2)$$

This is because the logged transactions are recovered (i.e., LT_j^k in line (1) above), but not those in a READY state. This will produce a different execution order than the one originally generated, that is, $LT_j^k \prec_{LH} GST_i^k$. Although either one is "serializable" the latter is unacceptable because of possible value dependencies. For example, assume that GST_i^k will become ready to commit only if the value of x is, say 1. Assume that $\hat{w}_j^k(x)$ wrote the value 10 to x. This results in GST_i^k either failing to become ready to commit (i.e. no pc_i^k exists) in the new history or possibly making a different decision, so the history could become:

$$LH^1 : \hat{w}_j^k(x); \hat{c}_j^k; r_i^k(x); w_i^k(x); a_i^k;$$

which is clearly unacceptable to the MDMS since it has already committed GT_i's operations.

When a DBMS provides strictness this cannot occur because other transaction are not allowed access to any other transaction's operations until after commitment. Finally, it follows that LRC histories are not permissible since ALCA $\subset$ LRC.

A second issue considered is the implementation of the

send ready to commit **accept** q_cond

line in each GST. The SYBASE and C example demonstrates the feasibility of such a communication technique. Note that the C function calls a communication procedure **MDMS_call** and waits for a response. A number of techniques could be envisioned to accomplish this inter-process communication ranging from a simple semaphore to a full inter-process communication protocol over large networks.

One additional comment is required to complete this section. During the interval that the ready to commit message is being processed the GST will maintain exclusive access right to data items. The local scheduler will block other transactions until the final commit is issued after the **send** returns the quit condition (q_cond).

5.3.4 Global Transaction Submission

The global recovery manager architecture depicted in Figure 5.3 is sufficient for the MDB two-phase commit approach. One additional point of a GT's execution must be logged. When a GST is ready to commit, it transmit a message to the MDMS which is logged in a GST ready to commit log (see Figure 5.5).

The ready to commit algorithm is presented in four parts emphasizing the logged information rather than those regarding timeouts and other issues. The first portion details the submission of GSTs that is information passed from the global scheduler to the global recovery manager. Three types of operations are passed between these: (a) B_GT to indicate the beginning of a new global transaction, (b) a termination condition (abort or commit) for each global transaction, and (c) the appropriately modified global subtransactions.

GT Management and GST Submission

1. The global scheduler transmits a B_GT operation to the global recovery manager. The GT identifier and the list of associated GST identifiers are recorded on the global log.
2. When a GST is submitted to the global recovery manager it is validated to ensure that a corresponding GT has been initiated. If the corresponding GT is not recorded, then the GST is rejected and the global scheduler is informed.
3. If the GST is valid, it is recorded in the GST active log on the global log. The GST is then submitted to the DBMS for servicing and the algorithm terminates. If the DBMS is not available the global recovery manager waits until the GST can be submitted. Other GSTs are not submitted by the global scheduler for this DBMS until the global recovery manager acknowledges the submission of the pending GST. If the DBMS is unavailable other subtransactions pending at other DBMSs can be processed.
4. When a GT is ready to commit or abort all required GSTs must be in a READY state. The global scheduler will be informed of this by the global recovery manager (to be discussed shortly in the second part of the algorithm). The decision to commit or abort is recorded in the GT Termination log. After recording the commit decision each DBMS which has a GST for that GT is

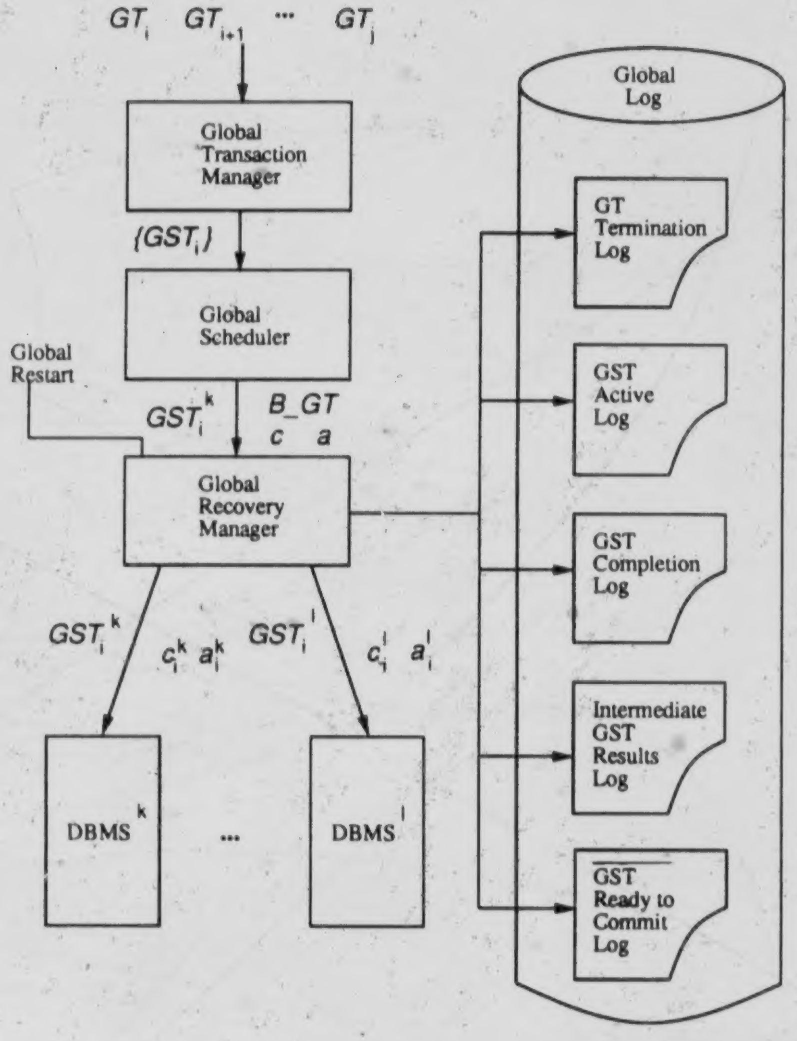

GT_i GT_{i+1} ... GT_j
Global Transaction Manager
$\{GST_i\}$
Global Scheduler
Global Restart
GST_i^k
B_GT
c a
Global Recovery Manager
GST_i^k
c_i^k a_i^k
GST_i^l
c_i^l a_i^l
DBMS^k
...
DBMS^l
Global Log
GT Termination Log
GST Active Log
GST Completion Log
Intermediate GST Results Log
$\overline{\text{GST}}$ Ready to Commit Log

Figure 5.5: Global Logging of the Global Recovery Manager

sent a message indicating the termination decision. If the DBMS is operating correctly the GST at the DBMS will commit or abort based on the decision. If the DBMS has failed while waiting for the final commitment decision the global recovery manager will wait for the DBMS to recover. During the recovery process the commitment decision will be passed to the GST. The protocol which accomplishes this will be described shortly.

5. The algorithm terminates.

The second process described relates to the ready to commit and final completion of GSTs at each DBMS. Each message is processed differently as illustrated below.

GST Ready to Commit and GT Completion

1. When a ready message is returned from GST_i^k at $DBMS^k$ the following steps are required.
    - 1.1 If the decision is to abort;
        - 1.1.1 The decision is returned to the global scheduler which informs the global recovery manager if the transaction should be resubmitted or if the corresponding GT_i should be aborted.
        - 1.1.2 If the GST is to be resubmitted:
            - 1.1.2.1 The GST active log is updated.
            - 1.1.2.2 The GST is resubmitted to $DBMS^k$.
        - 1.1.3 If the GT is to be aborted:
            - 1.1.3.1 The abort is logged in the GT termination log.
            - 1.1.3.2 Any GSTs which are READY for GT_i are aborted.
        - 1.1.4 If the GST_i^k is aborting a committed global transaction a DBMS failure has occurred which can be recovered from when the local restart process is executed[4].
    - 1.2 If GST_i^k is READY:
        - 1.2.1 The decision is logged in the GST ready to commit log.
        - 1.2.2 If GT_i has aborted GST_i^k is sent an abort message.
        - 1.2.3 If this is the last GST required by GT_i and all GSTs are READY a commit message will be sent. (See Step (4) of GT Initiation and GST Submission above.)

---

[4] This is performed in conjunction with the Global Handshake Process described in the next Section.

2. When a commit message c_i^k is returned from GST_i^k at $DBMS^k$ the following steps are required:

   2.1 The commitment is recorded in the GST completion log.

   2.2 The global scheduler is informed of the GSTs completion

   2.3 If this is the last GST to commit, GT_i is complete.

5.4 MDB Recovery Procedures

In the previous section atomic transaction commitment was discussed without considering failures. In this section, a recovery protocol which does not require DBMS modifications, but assumes that all DBMS are willing to cooperate with the MDMS in the recovery process is presented. By cooperation we mean that each DBMS recovers in conjunction with the MDMS. Once operational each DBMS makes all of its own decisions. The correctness of these algorithms is demonstrated in Section 5.5..

5.4.1 Modified Local Restart Process

Recall the stages involved in a local restart: the DBMS is started, recovery is performed based on the local log, and users are permitted access to the DBMS.

The local restart procedure can be altered, without violating local autonomy, since the restart is an external process. The modified restart is presented and a detailed discussion follows.

L-1. Restart the DBMS.

L-2. Recover the database by using information kept on the local log.

L-3. Open the database so that the MDMS has exclusive access.

L-4. Establish a handshake with the MDMS to notify it that the database is recovered. Wait until the MDMS responds and establishes communication.

L-5. The MDMS submits all GSTs that were ready to commit at the time of the failure. (This can be determined from the GST ready to commit log.) Note that all such GSTs will enter the READY state since each DBMS provides locally strict schedules.

L-6. The MDMS notifies the database administrator that exclusive access is no longer required.

L-7. The database administrator opens the DBMS for normal user access, that is, all other users are permitted access.

L-8. Terminate the local restart process.

Before discussing pragmatic details, we present the handshake at Step (L-4) and outline the information that the MDMS requires while the DBMS provides exclusive access. The MDMS must acquire the status of all GSTs at the time of the failure and resubmit any which were READY at the time.

G-1. Handshake initiated by DBMS is accepted.

G-2. Read the current status of all active GSTs from the global log for the DBMS which has initiated this communication.

G-3. Record the abort of any GSTs which were not READY.

G-4. Abort any GT corresponding to aborted GSTs from Step (G-3).

G-5. Record the commit of any GSTs which have committed and acquire any results required.

G-6. Resubmit all GSTs which were ready to commit at the time of the failure.

STARTUP NOMOUNT (a)
CONNECT INTERNAL (b)
ALTER DATABASE LDB^k MOUNT (c)

REVOKE CONNECT (d)
    FROM LOCAL_USER
ALTER DATABASE OPEN (e)

Handshake with MDMS (f)
MDMS Submits GSTs to READY state (g)
Handshake Complete (h)

GRANT CONNECT (i)
    TO LOCAL_USER

Normal DBMS Operations (j)

Figure 5.6: Modified Local Restart Procedure Using the ORACLE Syntax

G-7. Wait until all such GSTs have responded with a ready to commit message.

G-8. Terminate the handshake.

Each step of the Modified Local Restart Process is discussed and its feasibility illustrated using an existing DBMS. ORACLE[5] has been selected to illustrate the feasibility, however both SYBASE and INGRES[6] restart procedures are known to provide the necessary operations. The necessary ORACLE commands are presented in Figure 5.6.

Since we are describing DBMS restart procedures, we will begin by tracing the local restart operations and then trace the global handshake procedure when it is invoked. Step (L-1) and (L-2) are identical to the normal DBMS recovery steps outlined earlier. Line (a) and (b) of Figure 5.6 start ORACLE and inform the DBMS that the database is to be connected. Line (c) connects the database LDB^k and recovers LDB^k using the system log file. Currently users do not have access to LDB^k since the database is not OPENed.

---

[5] ORACLE is a registered trademark of Oracle Corporation.
[6] INGRES is a registered trademark of Relational Technology.

Step (L–3) requires the database administrator permit GSTs to access the DBMS without allowing other transactions access. Techniques for this will vary from one DBMS to another. Following recovery, it is possible to have the database administrator as the only user. Some systems require this, for example, SYBASE and INGRES. After the database is recovered, exclusive access is granted by revoking access to all users except the MDMS. We assume that two "users" exist on each DBMS, namely, the MDMS_USER and the LOCAL_USER. These could represent groups of users but the presentation is simplified by using only these two users. Line (d) of Figure 5.6 will REVOKE access to the local user and Line (e) opens the database. The requirements are met since the only user is MDMS_USER.

Step (L–4) and Step (G–1) establishes a connection with the MDMS using a handshaking paradigm. The DBMS is blocked until the MDMS communicates properly. This handshake notifies the MDMS that the DBMS has failed since it is possible that the MDMS was not aware of the failure. This occurs if no GSTs required access to the DBMS or those which had been submitted did not become ready.

Once the handshake is established, the MDMS determines the status of all outstanding GSTs (Step (G–2)) at the recovering DBMS. Any GSTs which were in their INITIAL state are moved to the ABORT state (Step (G–3)) and the corresponding GT is aborted. READY GSTs are submitted to the DBMS (Step (L–5) and Step (G–6)) and waits until they have all entered the READY state (Line (g) Figure 5.6). The MDMS performs all steps outlined in the Global Handshake Process. Once all such GSTs are READY, the MDMS informs the DBMS (Step (G–8), Step (L–6) and Line (h)).

Finally, the DBMS is opened to all users (Step (L–7)) by GRANTing access to LDB^k (Line (i) of Figure 5.6). The DBMS resumes normal operations (Line (j)) and the modified local restart procedure terminates (Step (L–8)).

5.4.2 Global Restart Process

One more process must be described. When the MDMS suffers a failure it is recovered using a Global Restart Process. As with DBMSs the operation is a collection of steps that must be performed by the MDMS's database administrator or a batch process capable of the same tasks. Recovery at the MDMS is straightforward. First, the MDMS is restarted. Secondly, the MDMS determines what happened during the failure. This requires the submission of a series of requests to each DBMS to determine the current status of global subtransactions active at the time of the failure. Once the condition of outstanding global subtransactions is known the global transaction log must be updated to reflect their new status. Finally, all other global subtransactions effected by the new status information must be updated.

Before presenting the algorithm we argue about the feasibility of retrieving the tus of global subtransactions at a DBMS and use this to motivate the algorithm. Assume, for the moment, that the DBMS is operational when the MDMS requests the information. First, consider the type of status information which can be returned by the DBMS when queried about active transactions. For example, SYBASE has a system procedure which returns the status of processes executing on an SQL Server called **sp_who**. This command will return the status of every active login, either sleeping or runnable, the loginname, host name, dbname, and command currently being executed. It is sufficient to know if a loginname corresponding to the GST being considered exists in the list returned by **sp_who**. To see this, consider the possible states a GST can be in when the MDMS failed.

INITIAL: In this case, the subtransaction is active at the time of the MDMS failure. If the subtransaction is still executing when the MDMS recovers it is permitted to continue. When it leaves the INITIAL state, it notifies the MDMS via the GST ready to commit and GT completion processes described in Section 5.3. If the GST is not in the list of active transactions then the MDMS knows that the GST has aborted, so the corresponding entries is made in the global log.

READY: subtransactions appearing in the global log in the READY state can be left in that state until the transaction termination decision is made. If a previously READY subtransaction is not still active, the MDMS assumes that the DBMS has failed and is waiting for the global subtransaction to be resubmitted during the global handshake procedure.

ABORT and COMMIT: Active subtransactions in these states are ignored since the final commit or abort commands are pending. If the subtransaction does not appear as an active transaction, it has completed normally.

Similar techniques exist on ORACLE for determining what users are currently logged onto the database and since this is sufficient we will present the global restart algorithm.

Global Restart Process

1. The MDMS is recovered via the operating system and global logs are restored, if necessary. Continue to accept responses from active global subtransactions but new global transactions are not accepted by the MDMS.
2. Determine the status of outstanding GSTs. For each active GST_i^k:
   - 2.1 Get GST_i^k status from $DBMS^k$.
     - 2.1.1 If the DBMS is not operational:
       - 2.1.1.1 Indicate that the DBMS failed.
       - 2.1.1.2 When communication is reestablished (later) perform Step (2) - (4) for all GSTs still outstanding at that DBMS.
   - 2.2 If GST_i^k is active at $DBMS^k$ then allow it to continue.
   - 2.3 If GST_i^k is committed:
     - 2.3.1 Record the completion on the GST complete log.
     - 2.3.2 Ensure that required termination results are recorded on the GST intermediate results log.
   - 2.4 If GST_i^k is aborted:
     - 2.4.1 If GST_i^k is not recorded in the GST ready to commit log then log GT_i as an aborted GT on the GT termination log.

   - 2.4.2 If GST_i^k is recorded in the GST ready to commit log then record GST_i^k's abort in the GST complete log. (Note that this is a DBMS failure.)

3. For each committed GT_i in the GT termination log:
   - 3.1 For each $GST_i^k \in GT_i$ we know it is at least ready to commit:
     - 3.1.1 If the most recent reference to GST_i^k is an abort in GST complete log the GST_i^k must be resubmitted to $DBMS^k$ with exclusive access since $DBMS^k$ failed. This will occur with a handshake when $DBMS^k$'s local restart is executed.
     - 3.1.2 If GST_i^k is committed it has completed correctly.
     - 3.1.3 If GST_i^k does not appear in the GST complete log the transaction is still active at $DBMS^k$ when its status was tested but had not yet completed its final commitment operation so it is allowed to complete.
4. For each aborted GT_i in the termination log:
   - 4.1 For each $GST_i^k \in GT_i$ which is READY:
     - 4.1.1 Communicate the abort of GST_i^k to $DBMS^k$, if necessary.
   - 4.2 When active GST's communicate with the MDMS they will return either a ready or abort message.
     - 4.2.1 If the GST is ready to commit, it is aborted and recorded in the appropriate log.
     - 4.2.2 If the GST is aborted, it will be recorded appropriately.

Step (1) is obvious since the underlying system must be running to enable the MDMS to become operational. Step (2) determines the status of all active GSTs at the time of the failure using the GSTs active log and the GSTs complete log. All incomplete GSTs in the GST active log must be checked since GSTs which are ready to commit and those that have not reached that state must be checked for status changes. Step (2.1) gets the status of each GST from each DBMS. If the DBMS is not functioning, no further processing is possible until the DBMS establishes a global handshake when it recovers. At that time, all GSTs will be recovered according to that recovery procedure. When the DBMS becomes operational it performs a handshake operation so status information can be attained and any GSTs can be resubmitted (Step (2.1.1.2)). Since other DBMSs may still be operational the global restart attempts to recover other GSTs (Step (2.1.1.3)). If the DBMS is operational, the possible responses are active, aborted, or committed. If the GST is still being processed it is permitted to continue (Step (2.2)). Since GSTs are permitted to continue normal operations during the global recovery operation the global recovery manager must accept responses from GSTs. These responses are recorded in the global log to ensure correct recovery.

If a GST committed (Step (2.3)) during the failure the commitment is logged in the GST Complete Log (Step (2.3.1)) and intermediate results must be saved in the GST intermediate results log (Step (2.3.2)). The actual mechanism that facilitates this depends upon the capabilities of each DBMS. If the MDMS can request previous results from the DBMS, then the results of a GST can be acquired during global recovery. If only the termination condition can be acquired another approach is necessary. The technique is to have the GST return both the ready message and intermediate results at the time the READY state is entered.

When a GST enters the READY state it passes all of the results to the MDMS. To see why this is necessary, consider a GST which is ready and has transmitted this information to the MDMS. Assume that the GT eventually commits so the global recovery manager responds to the **send** informing the GST to commit. Assume the MDMS fails between the reply telling the GST to commit and the final commit response from the DBMS. Further assume that when the MDMS recovers the only information retrievable from the DBMS is the fact that the GST committed but the values returned are lost. The global transaction manager cannot formulate a complete response for the global user and the GST cannot be resubmitted. To ensure that this does not occur, the GST returns information with the ready to commit message so the MDMS can save the results.

If the GST has aborted (Step (2.4)) then either the GST never entered the READY state or the DBMS suffered a failure during that of the MDMS. Each possibility is discussed. In the former case, the GST was never ready to commit so it and the corresponding GT are aborted. Subsequent steps will ensure that other GSTs for this GT are aborted as well. The latter case means that the DBMS failed while this GST was in the READY state. Since the corresponding GT has been committed it cannot be aborted retroactively. The GST must be resubmitted to the DBMS. The global recovery process accomplishes this by noting the abort in the GST complete log (Step (2.4.2)) and then finding committed GTs which have aborted GSTs in the complete log. These GSTs are resubmitted in the next step.

Step (3) searches the GT termination log to find active global transactions on the MDMS. Each GST associated with a committed GT must have been ready to commit before the MDMS failed so each GST must commit. If a GT's GST has committed it has operated correctly (Step (3.1.2)). If a GST has aborted, the corresponding DBMS must have failed with the GST in a READY state. Therefore, the GST must be resubmitted so that it can reenter the READY state and subsequently commit. This occurs when the DBMS provides exclusive access for the MDMS and all GSTs which were READY are resubmitted.

It is not immediately apparent that all global subtransaction can be submitted together when a recovering DBMS provides exclusive access. Submission of all subtransactions is possible because two conflicting GSTs cannot both be in the READY state at the same time. This can be illustrated by assuming that conflicting GSTs can be simultaneously READY. Consider the following history, which is a canonical example of two GSTs ready to commit:

$$LH^1 : r_1^1(x); w_1^1(x); pc_1^1; r_2^1(x); w_2^1(x); pc_2^1(x); c_1^1;$$

Assume that $DBMS^1$ failed after the pc_2^1 operation. If the history is permitted, the implied serialization is $GST_1^1 \prec_{LH^1} GST_2^1$. When a local restart receives both transactions together the local history could become:

$$LH^{1'} : r_2^1(x); w_2^1(x); pc_2^1; r_1^1(x); w_1^1(x); \cdots ; c_2^1;$$

$LH^{1'}$ serializes $GST_2^1 \prec_{LH^{1'}} GST_1^1$. If GST_1^1's pc_1^1 is value dependent upon x but the $w_2^1(x)$ altered x so that the value dependency fails, GST_1^1 would not be in the READY state as required by the global recovery operation. Fortunately, the history LH^1 is not possible since GST_2^1 has read from an uncommitted transaction, that is, $w_1^1(x) \prec_{LH^1} r_2^1(x) \prec_{LH^1} c_1^1$. Therefore, two global subtransactions can only be READY at the same time if the base sets are disjoint.

Step (4) searches the GT termination log to find global transactions which have aborted but still have outstanding GSTs. For each GST which is READY (Step (4.1)) the decision is sent to the GST if the DBMS did not fail during the MDMS failure or it does nothing (Step (4.12)). GSTs which are still processing will eventually report a ready to commit message or an abort. If the GST is READY, it is aborted (Step (4.2.1)) and the abort is logged. Aborted GSTs are logged.

5.5 Correctness of Two-Phase Commit Algorithms

The state diagram of Figure 5.4 is really identical to a two-phase commit state diagram except the MDMS (Coordinator) manipulates GTs while the DBMS (Participant) autonomously manages GSTs. Local transactions are managed at the DBMS without intervention from the MDMS, so they are not of interest to this discussion. Figure 5.7 illustrates the messages exchanged between the levels and the information logged by each system. The major differences between the traditional two-phase commit and the MDS implementation of the two-phase commit is in (1) the information logged when a transaction is ready to commit, and (2) the types of messages exchanged between the MDMS and the DBMS. Two-phase commit protocols will log when a transaction moves from the INITIAL state to the READY state. Traditional centralized DBMSs which do not have a two-phase commit move transactions directly from a processing state to a termination state. The MDB two-phase commit provides a READY state although the DBMSs themselves do not recognize the special state nor the need to log transactions when they enter it. Transactions move from state to state in the way depicted in Figure 5.7 where circles depict states and broken directed arcs depict the communication between the MDMS and a DBMS.

The balance of this section demonstrates the correctness of the multidatabase two-phase commit. We describe the termination protocols necessary when a response is expected but does not arrive, these are known as timeouts.

5.5.1 Termination Protocols

The termination protocols serve the timeouts of both the coordinator and the participant processes. A timeout occurs at the MDMS when it cannot get an expected message from a GST stub within the expected time period. Conversely, a GST stub times out when it does not receive an expected message from the MDMS. In this thesis, we assume that this is due to the failure of the transmitting MDMS.

The method for handling timeouts depends upon the timing of failures, as well as the types of failures. In Figure 5.7 any state expecting a message could cause a timeout. Therefore, timeouts can occur at the either the MDMS or at a DBMS. Each is discussed in turn below.

MDMS Timeout. There are three states in which the MDMS may timeout: WAIT, COMMIT and ABORT.

1. *Timeout in the WAIT state.* In the WAIT state, the MDMS is waiting for each DBMS to return a decision. The MDMS cannot unilaterally commit a transaction since any DBMS could unilaterally abort its global subtransaction *vis a vis* the corresponding global transaction. However, it can decide to globally abort the transaction by writing an abort record in the global log and notifying the operational DBMSs.

2. *Timeout in the COMMIT state.* In this case, the MDMS is uncertain that the commit operation was executed by the local recovery manager. If the DBMS

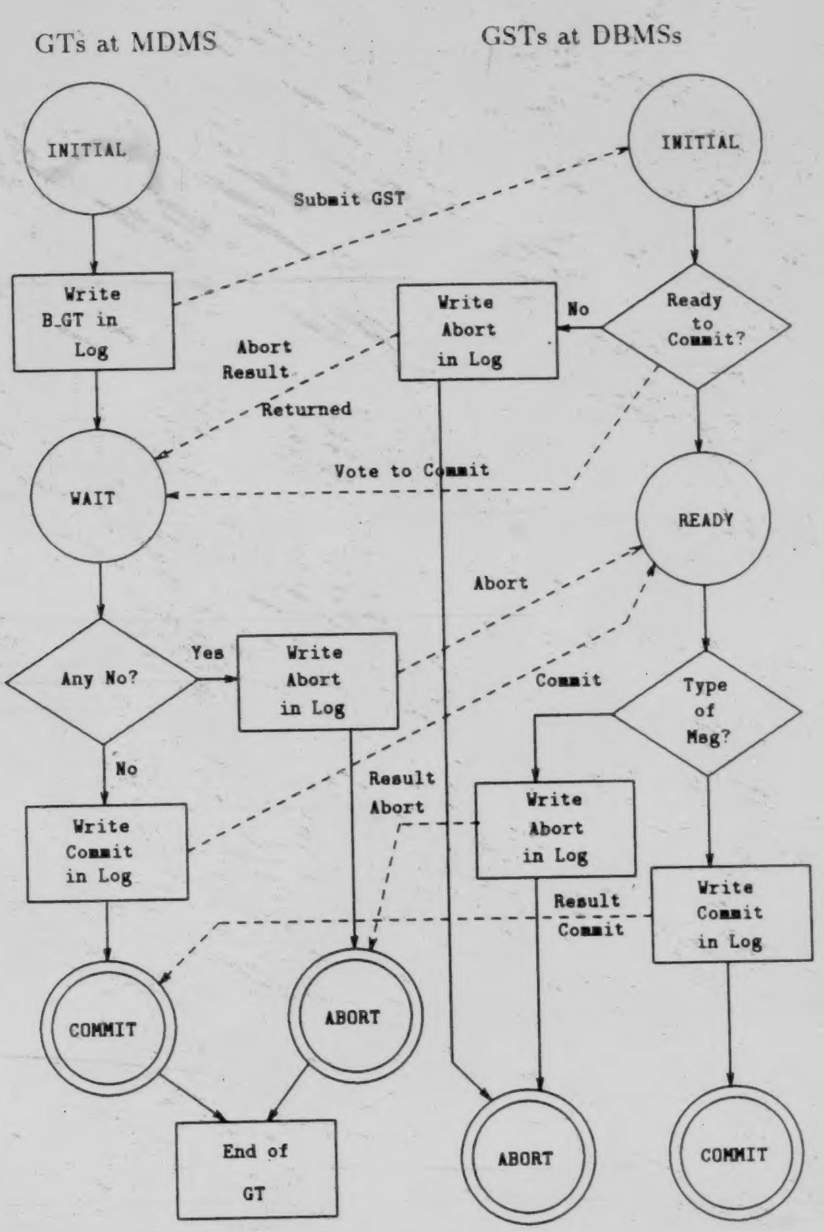

GTs at MDMS
GSTs at DBMSs
INITIAL
INITIAL
Submit GST
Write B_GT in Log
Write Abort in Log
No
Ready to Commit?
Abort Result Returned
WAIT
Vote to Commit
READY
Abort
Any No?
Yes
Write Abort in Log
Commit
Type of Msg?
No
Result Abort
Write Commit in Log
Write Abort in Log
Result Commit
Write Commit in Log
COMMIT
ABORT
End of GT
ABORT
COMMIT

Figure 5.7: State Diagram for MDB Two-Phase Commit Algorithms Including Logging

has failed and the decision was to COMMIT then the global transaction must resubmit the GST as a "new" one when the Modified Local Restart Process permits exclusive access.

3. *Timeout in the ABORT state.* If the DBMS failed and the decision was to abort then the global transaction and the corresponding global subtransaction have been aborted since the DBMS has no reference to the transaction. The global recovery manager must make a log notation.

DBMS Timeout. Note that a DBMS timeout must be embedded in the subtransaction itself since the DBMS does not know of the "special" nature of the subtransaction. A GST can only timeout in only the READY state. Since the GST is willing to commit it cannot unilaterally abort or commit at any time. Therefore, the GST remains blocked until receiving the commitment decision of the GT so embedded timeouts are useless.

5.6 Correctness of Recovery Protocols

In the previous section, we discussed how the two-phase commit protocol deals with failures from the perspective of the DBMS. In this section, we take the opposite viewpoint: we are interested in demonstrating that, given a MDMS (DBMS) failure the system recovers correctly. To function correctly following a failure each transaction must be in the same state after the failure as it was in before the failure, either committed and effect the databases or abort leaving all databases untouched. Numerous cases must be considered. Each case is presented by illustrating what occurs if a given system fails a transaction in any possible state and ensure that it is consistently recovered.

Actions taken on behalf of global subtransactions when a DBMS fails are described in Section 5.6.1. Actions taken for global transactions when the MDMS fails are described in Section 5.6.2. Finally, Section 5.6.1 argues that the two together can correctly recover from a failure of DBMS(s) and a failure of the MDMS.

5.6.1 DBMS System Failure(s)

DBMS failures can leave global transactions in any one of the four states. If a DBMS fails and the global recovery manager attempts to submit a GST to it (that is, the GT is in its INITIAL state) the GT is blocked as well as any others wishing access to that DBMS until local recovery has occurred (see Step (3) of GT Management and GST Submission). Information is not lost and inconsistencies are not introduced. If a GT is WAITing for a response from the DBMS, it will remain in that state until handshaking during Step (L-4) of the modified local restart process. Any GSTs which were READY are resubmitted, any other GSTs have aborted so the corresponding GT is aborted (see Step (L-5) of the modified local restart process and Step (G-3) and (G-4) of the global handshake process).

Global transactions in the ABORT state will not be effected by the DBMS failure. This is apparent since a GT's GST can only be in the INITIAL or READY state. Since both will abort during local recovery, the global recovery manager must simply log them as aborted (see Step (G-3) and (G-4) of the global handshake process). Any committed GTs must have their corresponding GSTs resubmitted with exclusive access, (Step (G-6) of the global handshake protocol) unless they are committed when the status is acquired, in which case the commitment is recorded and results acquired (Step (G-5) of global handshake process).

Global subtransactions can be in any of four possible states at the time of a failure. GSTs in the INITIAL state are aborted by the DBMS at recovery time. When local recovery occurs the status returned by the request at Step (2) of the global handshake process will be an abort. This results in the GST being logged as aborted and the corresponding GT being aborted. Global subtransactions which were READY will be reported as aborted by the DBMS when its status is requested. The MDMS resubmits the GST to the recovering DBMS in such a way that it is guaranteed to reach the READY state. The global restart process resubmits the GST at Step (5) and waits until all such GSTs have entered the READY state. All GSTs in the READY state will be in that state at the time the DBMS resumes normal operation, as required.

Global subtransactions which were in the ABORT state will not effect the database. GSTs in the ABORT state move to this state either by aborting unilaterally (which means that the GST moved from the INITIAL state to the ABORT state) or was READY at some point but was aborted by the MDMS. In either case, the consistency of the database is maintained since the GST has aborted and the MDMS is aware of the GST's status. Global subtransactions which were in the COMMIT state at the time of the failure will have the effects of their operations reflected on the database, as required. Since the GST is in the COMMIT state it must have passed through the READY state, so it follows that the MDMS knows the GST was ready to commit and the corresponding GT has committed.

Multiple DBMS failures requires that each recover autonomously in conjunction with the MDMS. Since each DBMS recovers GSTS in the same way, the global handshake process ensures that each DBMS arrives at the same decision.

5.6.2 MDMS System Failure

MDMS failure may leave global transactions in any state and may result in the failure of global subtransactions at the DBMSs. We discuss the status of the GSTs by describing which states they can be in when a GT is in a certain state.

Global transactions in the INITIAL state are aborted. This occurs because of the MDMS recovers its logs from the global stable storage at Step (1) of the global restart process. Global transactions in the WAIT state may also need to be aborted. GTs in the WAIT state have outstanding GSTs whose current status must be determined. The status is acquired at Step (2) of the global restart process. At this point, we will assume that the DBMSs are operational. Any global transaction which has an aborted outstanding GST is aborted at Step (G-4) of global restart process. Recall that a GST wishing to enter the READY state that cannot access the MDMS to inform it of its decision must abort, so only GSTs which are in the INITIAL state are still active. Obviously, WAITing GTs cannot have GSTs active which are in their COMMIT state.

Global transactions which were in the COMMIT state at the time of the failure may have outstanding GSTs in their READY, ABORT, or COMMIT state. As above, the status of outstanding GSTs is acquired. Those currently in the COMMIT state are logged (Step (3.1.2) of the global restart process), if necessary. If the GST's termination status is not recorded in the global log it is still active at the DBMS. If the GST has an abort entry in the global log the DBMS must have failed after the ready GST was sent a commit message but before the GST could committed. The GST must be resubmitted when exclusive access is obtained.

Global transaction which were in the ABORT state at the time of the failure could have GSTs in either the READY state or the ABORT state. If the GST is in the READY state the GST must be informed of the abort and the global log updated (Step (4.1) of the global restart process). Those GSTs which had aborted are recorded in the global log (Step (4.2) of the global restart process).

5.6.3 DBMS(s) and MDMS System Failure

Failures of both the MDMS and some number of DBMSs requires that each recover to a consistent state independently. Since it has been shown that each ~~system~~ is capable of recovering when only one system has failed and each recovers automatically, the only remaining issue is system synchronization when failure occurs at both levels.

Two possibilities exist.

When the DBMS recovers first, it attempts to establish a handshake with the MDMS at Step (L-4) of the modified local restart process. If the MDMS does not respond the DBMS is blocked. Once the MDMS recovers it will detect the outstanding handshake request at Step (2.1) of the global restart process. The information is acquired and logged in Step (2.2–2.3) of this process and the balance of the global recovery process is executed. Once normal processing at the MDMS can resume the global handshake process is performed. When all GSTs reach the required states the balance of the modified local restart process is executed.

Alternatively, if the MDMS recovers first, a "flag" is set to indicate the failed request for GST statuses at Step (2.1.1.2) of the global restart process. Once the DBMS attempts the handshake, the balance of the recovery can be completed, as above.

Chapter 6

Conclusion

Transaction management issues in multidatabase systems are studied in this thesis. The thesis has contributed in a number of important ways. Early research in the area suffered from the absence of a classification taxonomy. The first contribution of the research was to define, characterize, and classify multiple database systems thereby making it possible to identify the systems studied. These systems are described by a formal computational model which states assumptions explicitly thereby placing the research in its proper context.

Proper transaction management requires that the level of concurrency be maximized while maintaining reliability. Early research efforts in the multidatabase area utilized serializability as a correctness criterion for concurrent transaction execution. Unfortunately, the initial efforts did not produce schedulers because serializability is too restrictive. The thesis proposes a new, less restrictive correctness criterion called *MDB-serializability* which is an extension of the serializability theory. MDB-serializable execution histories form a superset of the serializable histories. The new theory captures the characteristics of both global and local transactions. Further, *multidatabase serialization graphs* (MSG) are developed to make it is easy to determine when a multidatabase history is MDB-serializable.

The new correctness criterion suggests schedules which do not violate DBMS autonomy. The thesis also contributes by describing two schedulers which produce MDB-serializable histories. The first scheduler is a conservative one that permits one global subtransaction to proceed if all of the global subtransactions can proceed. The "all or nothing" approach of this algorithm is simple, elegant, and correct. The correctness has been proven formally using MSGs. A second scheduler is proposed which is more aggressive, in that, it attempts to schedule as many of a global transaction's subtransactions as possible as quickly as possible. This requires more overhead to manage the global transactions but results in increased concurrency. The aggressive algorithm is proven correct by demonstrating that it creates the same class of histories produced by the conservative one.

The thesis also makes a very significant contribution in the area of reliability. Reliability is composed of two parts, namely, *transaction atomicity* and *recoverability*. The thesis contributes to the area of transaction atomicity by extending the two-phase commit algorithm to the multidatabase environment. Some of the advantages and disadvantages of the two-phase commitment approach in a multidatabase environment are discussed below.

MDB two-phase commit maintains DBMS autonomy since the ultimate GST commitment decision is made by each local recovery manager and no modification of the underlying DBMSs is necessary. Since a GST is permitted to abort, local value dependency checking is allowed with the ready to commit approach. Updates at multiple DBMSs are permitted, provided value dependencies do not span database boundaries. Finally, global subtransactions which are read-only can be structured so that the READY state is skipped thereby avoiding transactions blocking or holding locks on data item unduly long. In this case, the GST commits and the global recovery manager receives and saves the intermediate results in case they are needed if the GT commits.

Disadvantages to the approach exist. First, two-phase commit in combination with the computational model does not permit value dependencies that span DBMS boundaries. Since a GT can only submit a single GST at each DBMS, values at multiple DBMSs cannot be checked. Secondly, all DBMSs must provide a level of service which is at least locally strict.

An alternative approach that has been suggested is to force the commitment of any global transaction if any of its subtransactions commit. A number of different strategies have been suggested to force global subtransactions to commit. One approach is to submit subtransactions to the respective DBMSs and wait until they complete. If all of them commit locally, then the global transaction is considered committed. If one global subtransaction is rejected, then compensating transactions are sent to each DBMS to undo the effects of the committed subtransactions. Another alternative is to submit one subtransaction at a time and ensure that if the first subtransaction commits, all the subsequent ones will commit. This is accomplished by continually resubmitting rejected subtransactions until they commit. The major disadvantage to this approach involves forcing subtransactions to commit when the decision is made at an autonomous DBMS. It seems apparent that the approach would only work if some modification was made to the participating DBMSs or if the transaction model was changed so that a compensating transaction could "rollback" the effects of committed subtransactions. Our approach offers greater flexibility and makes no assumptions about the nature of the subtransactions beyond those imposed by a typical DBMS.

This thesis also contributes by describing recovery algorithms in the event of system failures. Recovery algorithms have been described that permit any system or combination of system failures including a single DBMS, multiple DBMSs, the MDMS, or the MDMS and DBMS(s). Arguments are provided which demonstrate the correctness of these algorithms.

Although this research has made a significant contribution in this area, a number of problems remain open. Based on this research new directions are suggested. The final few paragraphs of the thesis will state a few of the leading open problems but will not attempt to detail or describe possible solutions.

Formation of global subtransactions from a global transaction is an open problem. To date, significant research efforts have been directed at this area but in a different form. Previous work involved the formation of a global conceptual schema via schema integration. The corollary of the integration process is the formation and execution of queries suitable for each local DBMS. These research efforts have not yet been applied to the multidatabase environment.

A prototype design/implementation must be developed suitable for performing performance analysis. Both schedulers, on a simulated collection of DBMSs, should be implemented. Various DBMS characteristics can be implemented easily so that we can determine precisely when one scheduler will out perform the other. This analysis should provide guidance for scheduler implementation on a MDS and determine under what conditions a collection of DBMSs should be combined.

The distribution of the MDMS should be studied. Since the environment considered by this thesis is on a single site a number of complications can be ignored. Research into distributing the MDMS would make the work more generally applicable. Although no major difficulties are anticipated with distribution beyond those encountered in a distributed database system, the research will prove interesting.

Heterogeneity has not been considered in this work. Since the underlying DBMSs were assumed to be completely autonomous we were unable to take advantage of actual DBMS implementation strategies. For example, if the underlying DBMS utilized a timestamp approach it might be possible to acquire a number of sequential timestamps from the DBMS so the MDMS could force a certain serialization.

Inclusion of semantic information in the scheduler could increase concurrency. If the scheduler is able to detect the meaning of a transaction then it might be possible to submit more GSTs than the syntactic description indicates. Incorporating such semantic information would require fundamental changes to the computational model. There is already some work done in this area [31] but none that considers multidatabase systems.

Effects of relaxing the full autonomy assumption should be studied. A number of researchers have suggested this possibility [7, 16, 37]. Although this might be desirable, it might not be possible, since a DBMS is oftentimes proprietary software so source code is not generally available. However, if the restriction can be relaxed an increase in concurrency should be possible.

Finally, security in MDSs will become a leading practical consideration as these systems become a commercial reality. Local database administrators will be very reluctant to permit access to the MDMS if it is not properly secured. Issues such as inference and copy protection will become major concerns.

Bibliography

[1] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global proceedures in federated database systems. *Q. Bull. IEEE TC on Data Engineering*, 10(3):5–11, September 1987.

[2] K. Barker and M.T. Özsu. Concurrent transaction execution in multidatabase systems. In *Proceedings of COMPSAC'90*, Chicago, IL., October 1990.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, MA, 1987.

[4] Y. Breitbart, P. Olson, and G. Thompson. Database integration in a distributed heterogeneous database system. In *Proceeding of the 2nd International Conference on Data Engineering*, pages 301–310, Los Angeles, CA., February 1986.

[5] Y. Breitbart and P. Paolini. Session chairman overview – the multibase session. In F.A. Schreiber and W. Litwin, editors, *Distributed Data Sharing Systems*, pages 3–6. North-Holland, 1985.

[6] Y. Breitbart and A. Silberschatz. Multidatabase update issues. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data 88*, pages 135–142, June 1988.

[7] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. *Q. Bull. IEEE TC on Data Engineering*, 10(3):12–18, September 1987.

[8] Y. Breitbart, A. Silberschatz, and G. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data 90*, pages 215–224, 1990.

[9] A. Dogac and E.A. Ozkarahan. A generalized DBMS implementation on a database machine. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, pages 133–143, May 1980.

[10] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. *Proc. 13th International Conference on Very Large Databases*, pages 347–355, August 1989.

[11] W. Du, A. Elmagarmid, Y. Leu, and S. Ostermann. Effects of autonomy on global concurrency control in heterogeneous distributed database systems. *Proc. Second Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, pages 113–120, September 1989.

[12] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[13] A. Elmagarmid and W. Du. A paradigm for concurrency control in heterogeneous distributed database systems. In *Proceeding of the 6th International Conference on Data Engineering*, pages 37–46, Los Angeles, CA, February 1990.

[14] A. Elmagarmid and A. Helal. Heterogenous database systems. Technical Report TR-86-004, The Pennsylvania State University, University Park, Pennsylvania 16802, 1986.

[15] A Elmagarmid and A Helal. Supporting updates in heterogeneous distributed database systems. In *Proc. 4th International Conference on Data Engineering*, pages 564–569, February 1988.

[16] A. Elmagarmid and Y. Leu. An optimistic concurrency control algorithm for heterogeneous distributed database systems. *Q. Bull. IEEE TC on Data Engineering*, 10(3):26–32, September 1987.

[17] K.P. Eswaran, K.P. Gray, J.N. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[18] H. Garcia-Molina and K. Salem. Sagas. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data 87*, May 1987.

[19] D. Georgakopoulos and M. Rusinkiewicz. On serializability of multidatabase transactions through forced local conflicts. Technical Report UH-CS-90-20, University of Houston, 1989.

[20] D. Georgakopoulos and M. Rusinkiewicz. Transaction management in multidatabase systems. Technical Report UH-CS-89-20, University of Houston, 1989.

[21] V. Gligor and R. Popescu-Zeletin. Concurrency control isses in distributed heterogeneous database management systems. In F. Schreiber and W. Litwin, editors, *Distributed Data Sharing Systems*. Elsevier Science Publishers, B.V., 1985.

[22] V. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Information Systems*, 11(4):287–297, 1986.

[23] V.D. Gligor and G.L. Luckenbaugh. Interconnecting heterogeneous database management systems. *Computer*, 17(1):33–43, January 1984.

[24] J.N. Gray. Notes on database operating systems. In R. Bayer et al., editors, *Operating Systems: An Advanced Course*, volume 60, pages 394–481. Springer Verlag, New York, 1978.

[25] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 1:121–145, January 1988.

[26] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 2(2):253–278, July 1985.

[27] Y. Leu, A. Elmagarmid, and D. Mannai. A transaction management facility for InterBase. Technical Report Technical Report TR-88-064, Computer Engineering Program, Pennsylvania State University, May 1988.

[28] W. Litwin. An overview of the multidatabase system MRDSM. In *ACM National Conference*, Denver, October 1985.

[29] W. Litwin and A. Abdellatif. Multidatabase interoperability. *Computer*, 19(12):47–48, December 1986.

[30] W. Litwin and H. Tirri. Flexible concurrency control using value dates. In A. Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Database*, pages 144–149. IEEE Press, New York, N.Y., 1989.

[31] F. Manola. Applications of object-oriented database technology in knowledge-based integrated information systems. Paper prepared for the CRAI School on Recent Techniques for Integrating Heterogenous Databases, April 1989.

[32] E. Moss. *Nested Transactions.* The MIT Press, Cambridge, MA, 1985.

[33] M.T. Özsu and K. Barker. Architectural classification and transaction execution models of multidatabase systems. In *Proceedings International Conference in Computing and Information*, Niagara Falls, Canada, May 1990.

[34] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 1991 (in print).

[35] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MD, 1986.

[36] C. Pu. Superdatabases: Transactions across database boundaries. *Q. Bull. IEEE TC on Data Engineering*, 10(3):19–25, September 1987.

[37] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proc. 4th International Conference on Data Engineering*, pages 548–555, February 1988.

[38] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic locking: A stratagy for coping with long-lived transactions. Technical Report CS-TR-087-87, Department of Computer Science, Princeton University, April 1987.

[39] M. Stonebraker. *Readings in Database Systems.* Morgan-Kaufman Publishers, 1988.

[40] D. Tsichritzis and A. Klug. The ANSI/X3/SPARC DBMS framework report of the study group on database management systems. *Information Systems*, 1:173–191, 1978.